National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

NL 339 (r. 86/01)

196

0-315-26928-6

National Library
of Canada

Canadian Theses Division

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Division des thèses canadiennes

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

- Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur
PRASENJIT BISWAS

Date of Birth — Date de naissance

Country of Birth — Lieu de naissance
INDIA

Permanent Address — Résidence fixe
95/3A, HAZRA ROAD
CALCUTTA 700 026

Title of Thesis — Titre de la thèse
A CAPABILITY ARCHITECTURE FOR ADA

University — Université
UNIVERSITY OF ALBERTA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée
M. Sc.

Year this degree conferred — Année d'obtention de ce grade
1983

Name of Supervisor — Nom du directeur de thèse
John Tartar

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date
5th OCT. 1983

Signature
for P BISWAS

NL-91 (4/77)

THE UNIVERSITY OF ALBERTA

A Capability Architecture for ADA

by

Prasenjit Biswas

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

© Fall 1983

MICRO

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Prasenjit Biswas

TITLE OF THESIS A Capability Architecture for ADA

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall 1983

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) Prasenjit Biswas

PERMANENT ADDRESS:

95/3A HAZRA ROAD
CALCUTTA 700 026
INDIA

DATED 10th September 1983

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled A Capability Architecture for ADA submitted by Prasenjit Biswas in partial fulfilment of the requirements for the degree of Master of Science.

John Tartar
Supervisor
W. W. Armstrong
Keith A. Stromsmoe
Lee J. White

Date 10th September, 1983

Acknowledgements

I wish to thank Dr. Subrata Dasgupta for his enthusiasm and constructive criticism throughout the course of this work. I am also grateful to Dr. John Tartar for his help and encouragement in the later stages of this research.

Further thanks are due to the other members of my examining committee, Dr. W. Armstrong, Dr. Lee White and Dr. K. Stromsmoe for their helpful comments.

I must thank Rahman N. Farshi for many hours of interesting discussions and Ajit Singh for preparing some of the figures.

Finally, I am deeply grateful to my wife, Mitali, whose contribution in terms of encouragement, patience and understanding is immeasurable.

Abstract

This thesis embodies the design of a capability based stack processor best-suited for execution of programs in the newly proposed language Ada. Hence the principal emphasis in the thesis is on the structured development of a processor that reduces the semantic gap between the programs written in Ada and the object code produced by the processor.

One of the important features in Ada, that makes it different from other widely used programming languages, is its facilities for data abstraction. Moreover a highly desirable characteristic of a reliable computing environment is that it should support efficient execution of a process in a number of small protection domains implemented according to the 'principle of least privileges'. Both the above features are well supported in the proposed architecture through the definition of hardware recognised objects called packets and tagged capabilities.

Analysis of execution characteristics of any typical Ada program (on a conventional architecture) reveals that a considerable amount of execution time is spent in executing compiler generated code for Procedure Call-Return. Similarly a significant proportion of the execution time of an Ada process, executing in a protection system that is built around the 'principle of least privileges', is devoted to frequent domain switching. A new design methodology is proposed that facilitates the choice of efficient arcitectural support (hardware/firmware) for these two mechanisms. The methodology is based around the definition of a new complexity measure for exo-architectural components.

A considerable portion of the thesis is devoted to the description of the hardware organization of the processor and the associated capability mechanism. The choice of instruction forms and other architectural features are justified through critical analysis of proposals available in the related literature. In absence of any usage statistics for Ada, suitable statistics for other Algol like languages are used.

Table of Contents

Chapter Page

1. Introduction 1
    1.1 Language Ada and the Architecture 2
    1.2 Capability based addressing 8
    1.3 Organization of the thesis 13
2. Two Approaches to Capability Architecture Design 15
    2.1 The Partitioned Memory Approach 15
    2.2 The Tagged Memory Approach 16
    2.3 Domain Switching 17
    2.4 Tagged Architecture 23
3. Architectural support for Variable Addressing in Ada 26
    3.1 S*A Description and Virtual Transfer Complexity .31
    3.2 Variable Addressing Techniques 34
        3.2.1 Comparison of the AVTCs 57
    3.3 The Real Transfer Complexity 59
    3.4 Overall Performance 67
4. An Overview of the Proposed Architecture 73
    4.1 Memory Organization 75
    4.2 The Stack Processor 76
        4.2.1 The Salient Features and Problems in Stack Architecture 80
        4.2.2 A Preliminary View of the Stack Segment ...84
    4.3 The capability Mechanism 87
        4.3.1 The capability Representation 89
        4.3.2 Capability Mapping Mechanism 93
        4.3.3 The Set of capability Registers 94

4.3.4 Introduction of Domain Frames in the Process Stack 99

4.4 Some Special Issues in an Architecture for Ada .103

4.4.1 The Primitive Data Types 103

4.4.2 Support for Subranges and Constraint Checking 104

4.4.3 Support for Parameter Passing in Ada 108

4.4.4 Support for Dynamic arrays and Discriminant records 114

4.5 The packet Object 118

4.6 Implementation of Abstract Data Types 122

5. The Instruction Set 125

5.1 The Stack Group 125

5.2 The Capability Group 130

5.3 The Branch Group 132

5.4 The Control Group 133

5.5 The Array Group 135

5.6 The Arithmetic Logic Group 136

5.7 The Miscellaneous Group 136

6. Conclusion 138

Bibliography 142

List of Figures

Figure Page

1. Design Objectives 3

2.1 Domain Switching in Partitioned Memory 19

2.2 Domain Switching in Tagged Memory 20

3.1 A View of the Current Stack Frame 41

3.2 The Processor Organization 62

4.1 A View of the Stack Frame 86

4.2 The Capability Representation 92

4.3 Capability Register Format 95

4.4 A Modified View of the Process Stack 100

4.5 Integer Words 105

4.6 A Rangeword 106

4.7 A Procedure Activation Record 113

4.8 Array Representation 116

4.9 Representation of a Packet 121

Chapter 1

Introduction

In recent years, many efforts are being made to develop appropriate language and system supports for large scale software development/maintenance and reliable software. The complexity of any large system is more manageable when it is decomposed into relatively *stable* subsystems. This was suggested by Simon [Sim69] as a common organizing principle of all complex systems. In the domain of software engineering, similar notions of hierarchical complexity decomposition have been advocated through the introduction of the concepts of structured programming, program modularity and information hiding. [Dij68, Par72]. Various programming languages - MODULA, ALPHARD, CLU, etc., have been proposed to consolidate these concepts [Hor83]. The design of programming language Ada could be considered as a culmination of all these efforts. One of the prime objectives for the design of Ada was to propose a language that offers significant advantage in large scale software development. It provides excellent features for data abstraction and modularity.

It is well understood that two of the major requirements for implementation of reliable software are [Lin76]: (a) execution of processes in small protection domains and (b) system structure to support implementation of the concept of abstract data types and information hiding.

This thesis embodies the design of an architecture that has the following objectives:

a. provide architectural support for efficient execution of programs written in Ada,

b. provide architectural support for elegant implementation of the concept of data abstraction and modularity as in Ada,

c. provide support for implementing the principle of 'least privileges'[1] (small protection domains) and flexible sharing.

These architectural features facilitate development and implementation of large scale reliable software. The objective of the proposed design is represented in Fig 1.1 (the arrows between the blocks are to be read as "supports"/"facilitates").

1.1 Language Ada and the Architecture

A review of the research efforts in languageoriented architecture design reveals that the primary aim of such efforts is to reduce the semantic gap between the high-level language (HLL) and the architecture that would execute programs written in the language. Such architectures are generally categorized into the classes of high-level language architectures and language-directed architectures[2] [Mye82].

---

[1] This concept is introduced in section 1.2.

[2] These terms have no universally accepted definitions.

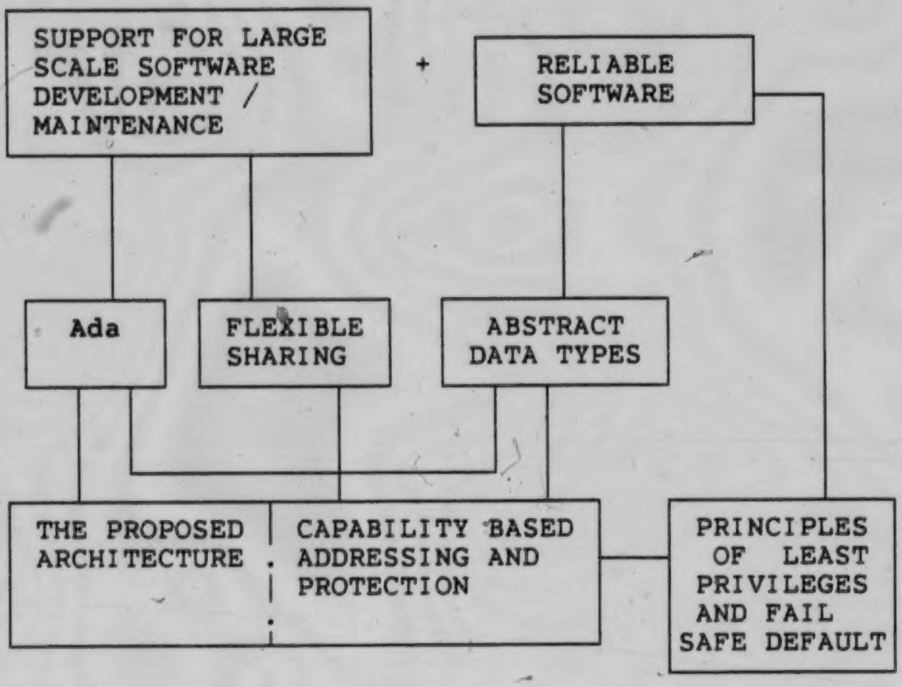

SUPPORT FOR LARGE SCALE SOFTWARE DEVELOPMENT / MAINTENANCE
+
RELIABLE SOFTWARE
Ada
FLEXIBLE SHARING
ABSTRACT DATA TYPES
THE PROPOSED ARCHITECTURE
CAPABILITY BASED ADDRESSING AND PROTECTION
PRINCIPLES OF LEAST PRIVILEGES AND FAIL SAFE DEFAULT

Figure 1.1. Design Objectives

The characteristic feature of high-level language architecture is that the high-level language is either considered as the assembly language of the machine or interpreted directly by microcode or hardware. These architectures practically reduce the semantic gap to zero but have severe disadvantages and limitations that make them practically infeasible [Mye82, DiP80].

The language-directed approach to architecture design implies an exo-architecture[3] [Das82b] that is designed with high-level languages in mind such that this thinking permeates all of the architectural design decisions [Mye82]. The operations and data structures frequently used by the programs written in the HLL are usually supported by semantically equivalent exo-architectural features. Essentially an attempt is made to evenly distribute the complexity of the HLL program to machine language mapping, between the compiler and the interpretive mechanism of the architecture (microprogram or hardware). The design proposed in this thesis is an architecture directed towards Ada.

As far as architectural support is concerned, Ada is another block-structured Algol-like language with additional facilities of modularization and data abstraction. In most of the previous Algol-like languages, procedure was the primary abstraction mechanism. The data abstraction in a programming language is a mechanism which encapsulates the representation and allowed operations of a data type. The representation of the type remains invisible to the user. An object of the type is allowed to be manipulated only by the operation specified by the implementor and specified within the encapsulation. Ada defines an abstract data type through the use of Package modules and Private type declarations [GoH81].

---

[3] The architecture as viewed by the compiler writer.

Numerous proposals have been made for language-directed or HLL architectures for block-structured languages [Mye82]. Among the commercial architectures, a series of machines from Burroughs [Dor79, Org73], MU5, ICL2900 [BiB80] series of machines (to name a few) could be considered to be directed towards Algol-like languages. So the obvious question arises: what are the new features in the proposed architecture that makes it distinct from the earlier designs? The following paragraphs would be indicative of some of the distinctive features of the architecture.

A feature that distinguishes Algol-like languages from other procedural languages is the mechanism of variable addressing that implements the notion of scope/visibility of free variables in these languages. Various techniques for implementing this mechanism at the compilation level have been proposed in the literature. One of the earliest techniques for implementation of variable addressing mechanism is due to Dijkstra [Dij60] and one of the latest one is due to Tanenbaum [Tan78]. Until recently [Dep82a] no formal performance analysis of these proposals was reported in the literature. Some of the architectures have provided support for variable addressing in block-structured languages but designers did not provide any formal argument for the choice of the architectural support. Only Tanenbaum informally justified the support in his proposal [Tan78] by relating it to usage statistics of the language under consideration.

A new methodology is proposed in this thesis that not only provides a formal basis for performance analysis of these mechanisms but directly indicates the most suitable architectural support for the execution of the block-structured language under consideration. The two new complexity measures - virtual transfer complexity and real transfer complexity, proposed in this thesis, provide the basis for the methodology.

It should be noted that a significant amount of execution time is spent in management of the block-structured environment for enforcing the scope/visibility rules of these languages. The choice of the variable addressing mechanism directly influences the complexity of implementation of *Pre Call, Pre entry, Post return and Post exit* sequences necessary for the maintenance of the block-structured environment. The high frequency of usage of procedure call-return and block entry-exit in structured programs is well known [Dep82b, Tan78, Mye82]. It should be obvious from the above discussion that the design of the architectural support for variable addressing plays a key role in determining the efficiency of a language-directed architecture for a block-structured language like Ada. The above mentioned methodology has been used in the design of the architectural feature for the variable addressing mechanism in the proposed design.

Ada supports separate compilation of program units to facilitate large scale software development. The program units in Ada are - subprograms, packages and tasks[GoH81]. In this proposal, the storage is not viewed as a linear sequence of words; rather it is viewed as a set of objects. The word 'object' signifies a group of related storage elements with the same lifetime. A similar view of the storage has been adopted in many of the recent proposals for new architectures [Mye82]. The compiled version of the above mentioned program units are represented at the architectural level as hardwire/firmware recognized objects called *Packets*. A packet can encapsulate the compiled version of a package, one or more subprograms. The packet has some similarities with the module object in the SWARD architecture [Mye82]. A closer look at the functional characteristics reveals the superiority and the distinctive features of the proposed object in the context of Ada. Architectural support for multitasking has not been included in this preliminary version of the design.

Some of the new features in Ada that are supported by the proposed design are indicated below.

1. Ada provides facilities for data abstraction through the

use of packages and private type declarations. The proposed architecture provides a new and elegant mechanism for implementation of abstract data types.

2. The language allows dynamic arrays and discriminant records. The actual representation of these composite structures might not be known until the execution time. The architecture has adequate features for representation and manipulation of such objects (which includes support for run time subscript bound checking).
3. Ada allows declaration of subtypes and definition of subranges. Furthermore subranges could be determined at run time. The architecture introduces new primitive data types and instructions that allow efficient implementation of run time constraint checking.
4. The semantics of the parameter mechanism in Ada and the necessity of dynamic type checking pose a new problem for the implementors of the language. The proposed design incorporates an unconventional but efficient feature in the architecture that facilitates the handling of Ada parameters.

1.2 Capability based addressing

The concepts of 'capability' and capability based protection were introduced by Dennis and VanHorn in their classic paper on "Programming Semantics for Multiprogrammed Computations"[DeV66]. A capability is a pair (x,r) specifying the unique name (logical address) of an object x and set of access rights (eg., read data, write data, read capability, enter etc.) for x [Den82, Mye82, Lin75, SaS75]. The capability is a ticket in a sense that possession of the capability unconditionally authorizes the holder r-access to x. Various systems have used capabilities in quite different ways, but a capability representation would generally have the following attributes:

a. a capability identifier, representing a system-wide unique name for an object (often the identifier has been loosely used as a 'capability' in the literature) and

b. a set of access rights that the capability allows to the object that it names.

As mentioned earlier, the storage in this architecture is viewed as a set of objects. These objects are created via machine instructions and are named at the time of creation. The names are returned to the creating process. These names/ identifiers represent logical addresses of the objects. The term capability as used in this thesis represents an occurrence of these names. Moreover it is ensured that these names are system-wide unique and are never reused in the lifetime of the system.

The principle of capability based protection implies that a process could access an object if and only if it has the 'capability' for the object with appropriate access rights. This leads to the notion of capability based addressing [Fab74]. In a system using capability based addressing, a table is maintained in the system that contains the information required to translate a logical address (capability identifier) to a physical address in the primary memory. When a system integrates capabilities into the hardware for memory addressing mechanism, the architecture is generally categorized as a 'capability architecture'. In a capability architecture a capability is interpreted on each reference to primary memory.

A protection domain is an independent local address space defining the total set of addresses that can be formulated by a set of instructions [Mye82]. The well-known 'principle of least privileges' propounded as a desirable characteristic for protection models for secure and reliable computation indicates that a program should have access to only those objects that are necessary for successful execution of the program [DeV66, SaS75, Lin76, Den82, Mye82]. This principle dictates that a process should execute in a number of small protection domains. Usually a protection domain is associated with a protected procedure [DeV66, Lin76, GrD72] and a process is executed by calling these protected procedures. The instruction representing a call to such a procedure, during the execution of the process, is referred to as the 'Enter' instruction in the literature. The execution of an 'Enter' instruction causes switching of protection domains. In a system using capability based addressing and protection, a protection domain is characterized by a set of capabilities.

The principal motivation in choosing capability based addressing as the basic addressing mechanism in the proposed architecture was to provide efficient architectural support for implementation of packages and abstract data types in Ada. In addition to providing this support, the use of capabilities and capability based addressing facilitates flexible sharing and run time implementation of 'the principle of least privileges'[GrD72, Fab74, SaS75, Lin76, Mye82].

In this proposal a protection domain is associated with a packet. The granularity of protection might be considered to be coarser than most of the capability architectures proposed earlier (e.g., system 250, IAPX 432, Cambridge CAP computer etc). It introduces the notion of 'user controlled granularity of protection', in the sense that a packet object could represent one or more than one subprograms. The finer granularity could be easily achieved if each and every subprogram is compiled separately and thus would get represented as separate packets. This option is provided to facilitate efficient execution of Ada programs.

Capabilities have obvious similarities with the segment descriptors used in systems using segmented memory management. The principal difference is that capabilities represent systemwide unique names and handling of capabilities does not lie with any privileged state of the system. Thus capabilities may be easily passed between protection domains, providing flexible but controlled sharing of data and procedures. The use of capability based addressing as an uniform method of addressing shared objects has been ably demonstrated by Fabry [Fab74].

An important requirement for implementing capability based addressing and protection is that there should be some basic mechanism in the architecture that distinguishes capabilities from data. There are essentially two approaches to provide this distinction. A brief explanation of these two approaches for capability based architecture design is given in Chapter 2.

In this architecture the capabilities are considered as a primitive data type in the machine and the definition of the architecture prevents any user program to fabricate capabilities. Every object in the architecture is addressed through a capability. The objects in the architecture that are protected through the capability mechanism are: the process stack segment, packets and objects created in the heap.

One common objection to the use of capability based architectures is that if every address is resolved through a capability, the additional indirection implied in the scheme leads to inefficient execution. It is almost universally accepted that a system that allows implementation of principles of 'least privileges' and 'fail safe default' (i.e., access based on explicit authorization) provides a more secure and reliable computing environment [Sas76, Den80, Den82, Mye82]. A system using a capability based protection mechanism would easily allow the implementation of these policies. But the execution time overhead implied in frequent domain switching makes these architectures unattractive. The proposals of architectures that use tagged capabilities [Mye82, Den80, Jag80, Wil72] significantly reduce this overhead (explained in chapter 2). The use of special registers to facilitate capability addressing [Eng74, Den80] leads to reduction of the overhead of indirect addressing. The proposed architecture uses capability registers and tagged capabilities.

1.3 Organization of the thesis

The next chapter includes discussion on the two major approaches to representation and handling of capabilities in the existing capability architectures. It also contains a brief discussion on the advantages of the tagged capability representation and 'self-identifying' data (in general).

Chapter 3 deals with the issue of architectural support for variable addressing in Ada. Two new complexity measures for exo-architectural components are developed and the measures are used as the basis of a design methodology for designing architectural support for variable addressing. The methodology is finally used in choosing the architectural support for variable addressing in the proposed design.

Chapter 4 describes the proposed design. All the distinctive features in the design are explained in separate subsections. The rationale for choosing a stack oriented architecture for Ada is also presented in this chapter.

A summary of the semantics of the instructions proposed in this architecture is provided in Chapter 5.

Finally, the last chapter discusses the results of this research and includes some suggestions for further study. A few critical remarks on two existing architectures (SWARD and IAPX 432), in the context of execution of Ada programs, are also provided.

Chapter 2

Two Approaches to Capability Architecture Design

The integrity of capability based addressing or protection mechanism totally depends on the protection of the capabilities. It is important to distinguish between the capability information and data stored in memory. Essentially there are two approaches to protection of capabilities. These two methods distinguish the two approaches to capability architecture design. The two approaches are:

a. the partitioned memory approach

b. the tagged memory approach

2.1 The Partitioned Memory Approach

The partitioned memory approach is a natural extension of the virtual memory mechanism. In virtual memory systems using segment descriptors, the segment descriptors are stored in segments that are only accessible to the supervisor state of the machine. In the partitioned memory approach to capability architecture design, capability and data information are stored in different **types** of segments. The data words and capability information are never allowed to reside in the same segment. Some of the architectures designed using this approach are the IAPX432 [RaL81,Mye82], the Cambridge CAP[NeW77] and the System 250 [Eng72].

The capabilities are stored in segments commonly referred to as capability segments, C-lists or access segments. All the references made by a program are interpreted indirectly through a current C-list. As mentioned earlier, a C-list or access segment is associated with a domain of protection. The capabilities stored in the C-list determine the objects that could be addressed by the program referring the list. The change of current C-list is equivalent to a domain switch operation. In the partitioned memory approach the capabilities serve the purpose of information indentification over and above addressing and access control.

The capability identifies the information in the object it names, through the kind of access it allows to the object. Usually access rights present in a capability could be data rights or capability rights. A capability can not have both the rights to the same object.

2.2 The Tagged Memory Approach

In this approach the capabilities are distinguished from other information through tags provided in every word of memory. Thus in this representation every word in the memory is 'self identifying' [Feu73]. The capabilities are protected at the word level by a specific tag that distinguishes a capability from other information and allows specific instructions to use the capability information. Therefore mixed segments containing both data words and capability words are allowed in the architectures designed with this approach. Some of the architectures of this kind are - the IBM System 38, and the IBM SWARD.

2.3 Domain Switching

As mentioned in Chapter 1, efficient domain switching is of utmost importance in capability architectures. The operation of domain switching in the two approaches will be examined in this section. The comparison is important in the sense that the efficiency of domain switching dictates the choice of a particular approach of capability architecture design for the proposed architecture.

Each object in a partitioned memory machine is represented by at least two segments, one containing the data and one containing capabilities. In a tagged machine the data words and the capabilities can be stored in the same segment. This feature directly influences the number of segments involved in the representation of a process. In the tagged approach, the number of segments required will be less and thus fewer capabilities will be required to represent the domain of a process.

In a partitioned memory approach, a domain switch involves changing of two segments (e.g., consider the ENTER instruction, as proposed originally by Dennis and VanHorn [DeV66]):

(i) change of segments representing the code, i.e., a new code segment will be entered by executing the ENTER instruction; and

(ii) change of C-lists for changing of protection domains.

The mechanism is explained with the figures 2.1(a) and (b). The figure 2.1(a) depicts the state before execution of the ENTER instruction and the figure 2.1(b) represents the state after execution of the ENTER instruction. in figure 2.1(a) the process is considered to be executing in the protected procedure represented by the segment P1, before executing the Enter instruction. The execution of the Enter instruction causes switching of protection domain and the process executes in the protected procedure represented by the segment P2. In figure 2.1(a) the program counter(PC) specifies the capability for the code segment P1 and the offset in the code segment containing the Enter instruction. The ENTER instruction specifies the capability pointing to the new C-list2 and also specifies the offset in the new C-list (containing the capability for the new code segment). A domain pointer DP, shown in the figure, points to the C-list representing the current protection domain.

In a tagged machine, the ENTER instruction may only need a change of code segments to enter a new procedure, since the capabilities required by procedures can be embedded in the code segment. The protection domain could be represented by the capabilities present in the code segment. The domain change operation in tagged capability architectures is shown in fig 2.2. Figure 2.2(a) represents the state before execution of the Enter

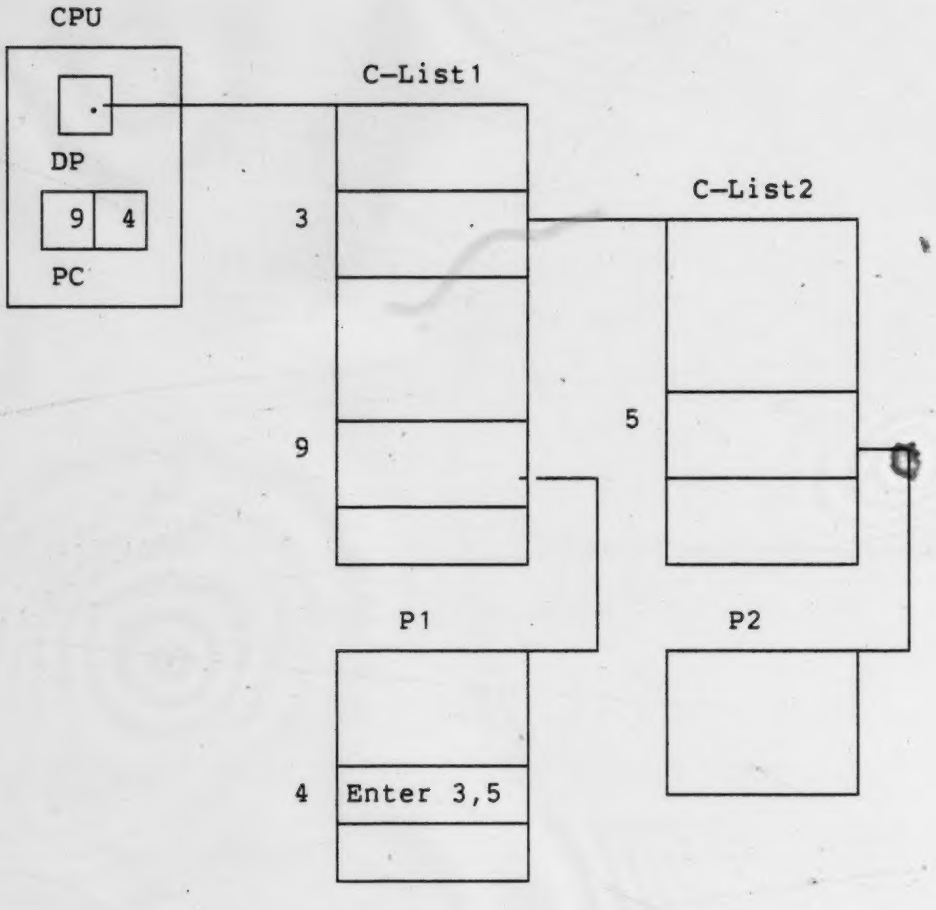

CPU
DP
9
4
PC
C-List1
3
9
C-List2
5
P1
4
Enter 3,5
P2

Figure 2.1(a)

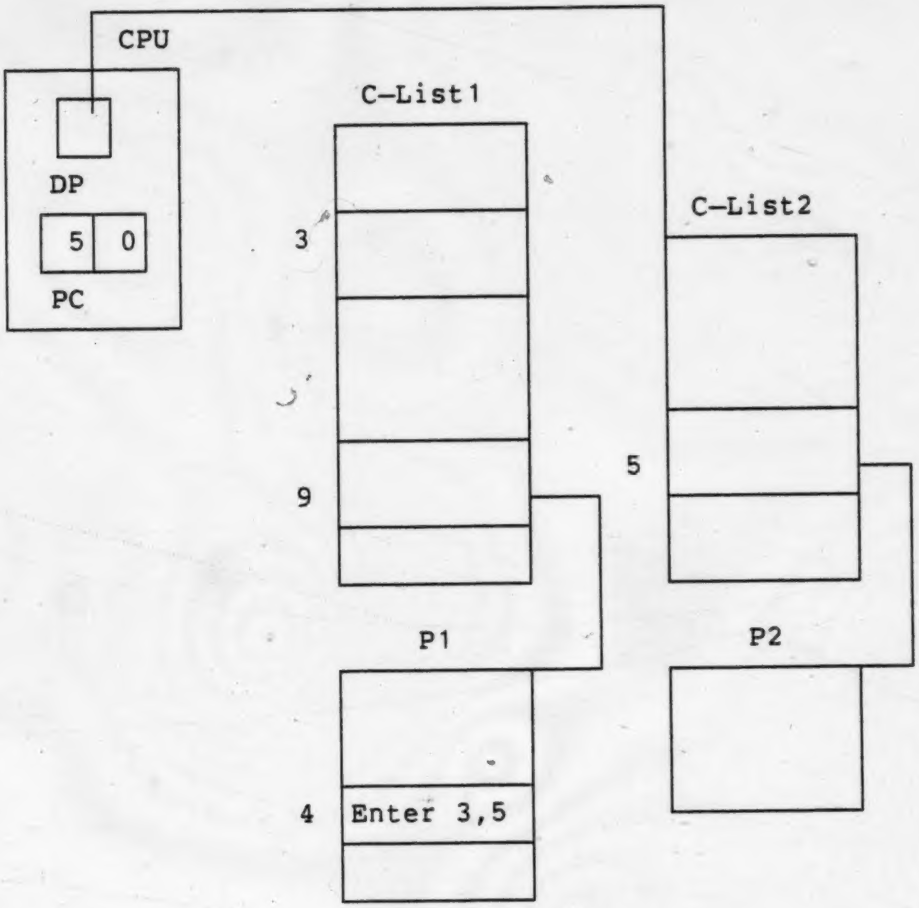

CPU
DP
5
0
PC
C-List1
3
9
C-List2
5
P1
4
Enter 3,5
P2

Figure 2.1(b)

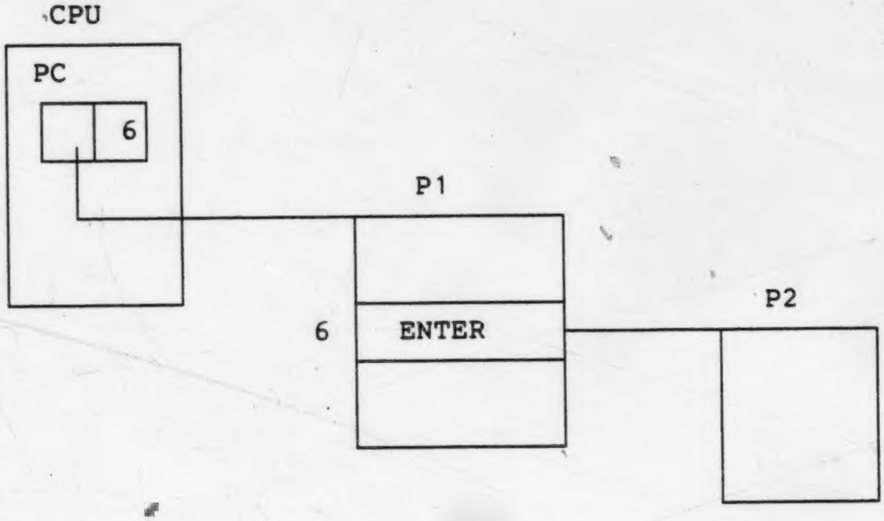

CPU
PC
6
P1
6
ENTER
P2

Figure 2.2(a)

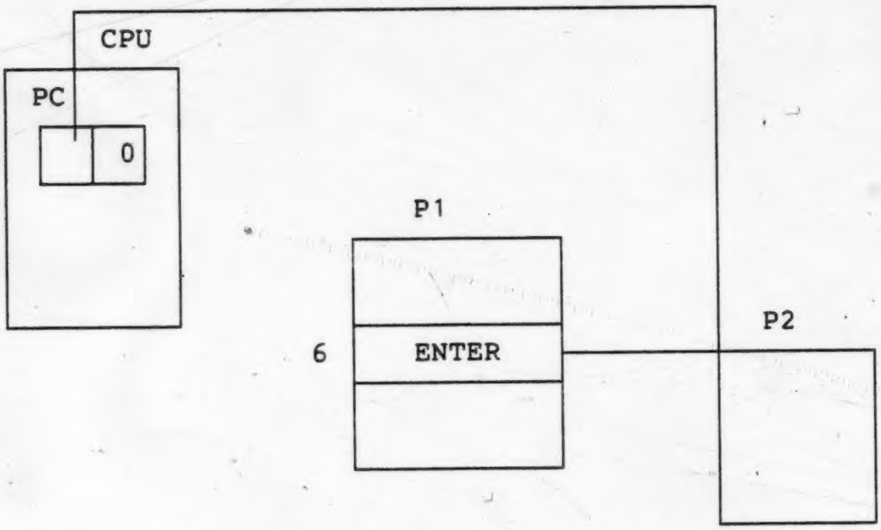

CPU
PC
0
P1
6
ENTER
P2

Figure 2.2(b)

instruction and figure 2.2(b) depicts the state after execution of the Enter instruction. The program counter represents an address in the segment number, offset form.

Moreover it should be noted that the addressing mechanism in a tagged capability architecture is much simpler than the architectures based on the partitioned memory approach. To address a word using capability based addressing, a capability for a segment and the offset of the word relative to the base of the segment must be specified. In the partitioned memory approach, these two components of the address can not be stored together in the same segment, as the offset is represented as a data word or as a portion of an instruction word. For example, an address to a data word is specified as a triple (i,j,k) in the CAP Computer [NeW74]. The component i selects one of 16 capability segments, j selects a capability from the segment and k specifies the offset. This extra level of indirection required to specify a capability in one of the capability segments is not required in tagged architectures. An address could be directly specified by a (capability, offset) pair.

The above discussion indicates that tagged capability architecture allows faster domain switching and operand addressing. These were the two factors that were considered for adopting the tagged capability approach in the proposed design.

2.4 Tagged Architecture

The tagged capability approach to capability architecture design implies that each and every word of the memory would have a tag field identifying the word. It is important to analyse whether the tagged memory approach fits in with the other objectives of the proposed architecture specifically oriented towards Ada.

The uses and advantages of tagged architectures have been adequately demonstrated in the literature [Feu73,Mye82]. Some of the salient features of tagged architectures are summarized below:

a. In a tagged architecture the data representation is self identifiying. The self identifying nature of data allows use of generic instructions. The architecture defines generic instructions, thus the instruction set is simpler.

b. Tagged architectures allow automatic data conversion. It has been shown by Myers that a large portion of the execution time is spent in executing compiler generated code for data conversion in non-tagged architectures.

c. In tagged architectures the binding of instruction to data attributes is deferred till the execution time. This provides execution speed benefit for languages requiring this feature of late binding.

d. Type checking at execution time is automatically done in tagged architectures (the necessary sequences for type checking are embedded in the microcode/hardware). This feature eliminates the necessity of compiler generated code for run time type checking.

There are two approaches to the design of the tag field - long tags and short tags. The long tag approach includes type as well as descriptor information in the tag field. The usefulness of long tag fields have been demonstrated by Gehringer [Geh79] and Myers [Mye82]. The type identifier part of the tag specifies the format of the descriptor and the descriptor part defines the format of the contents of the 'cell'. A cell can be arbitrary expanse of memory. The long tag approach allows representation of arbitrary cell types and facilitates implementation of the run time structures for languages having execution time binding features.

Ada is a strongly typed language [Hor83]. Excepting for the few cases (mentioned in Chapter 1), the type compatibility of operands could be checked at compile time. Thus use of long tags will not provide any execution time benefits for Ada programs. As mentioned earlier, Ada requires execution time support for run time range-constraint checking and bound checking for array subscripts.

The proposed design uses 4 bit tags. The rationale behind the design of the tag field is explained in Chapter 4. The primary motivation for using tagged approach is hardware protection of primitive data types in the architecture (quite similar to the rationale for tagging in 5700, 6700 and 7700 series of machines from Burroughs [Dor79]).

Chapter 3

Architectural support for Variable Addressing in Ada

The importance of usage statistics of various high level language (HLL) constructs in the design of the instruction set of a language directed machine or the execution architecture for the HLL is well understood [Mye82, FlH83]. The choice of the *technique* used in interpreting the directly-executable language (DEL)[FlH83] constructs or the instructions of a language-directed architecture thoroughly influences the performance. It is interesting to note that no well established design methodology for choosing the interpretive mechanism or for determining the hardware/firmware support for an efficient implementation of the interpretive mechanism is not known. The research presented in this chapter might be considered as an attempt towards formulation of such a design aid.

Ada is the HLL under consideration in this thesis. It is one of those HLL's where the variable referencing environment is determined by the static block structure of the programs [Hor 83]. Scope rule enforcement for the variable access mechanism represents a characteristic feature of such languages. The choice of the technique for implementing the variable addressing mechanism directly affects the complexity of the Pre Call/Pre Entry & Post Return/Post Exit (Entry and Exit correspond to Block Entry & Exit respectively) sequences. These sequences are not directly involved in variable addressing but they are essential overhead for maintenance of the mechanism.

It is well known that procedure Call/Return & Block Entry/Exit are two of the important and frequently used operations in a block-structured HLL environment. So they are definitely candidates for having semantically equivalent constructs in the execution architecture for the HLL or instruction set of language -directed machine. The one to one correspondence of such functions in the instruction set leads to effective use of the available processor/memory bandwidth [Mye82]. This in turn implies that an instruction like Call (for example) includes the firmware implementation or hardware control for the Pre Call sequence over and above the usual microcode or hardwired sequences for transfer of control. The execution performance of such instructions will depend on the *technique* used to implement the variable addressing mechanism. The designer of a language directed architecture (or the interpreting mechanism for a DEL) is faced with the problem of correctly choosing the most efficient implementation technique. The discussion to follow is quite different from DePrycker's analysis [Dep82a], in the sense that he has attempted to evaluate two different variable addressing methods on existing target architectures. The emphasis in this chapter is primarily on the development of a methodology for designing the hardware/firmware support for such HLL functions for a language directed architecture.

Various techniques have been used by system designers to implement the variable addressing mechanism in block structured languages. Four competitive techniques that have appeared in the literature are considered. The techniques to be considered are:

a. the classical display implementation as suggested by Dijkstra [Dij60, Dep82];

b. a modified display implementation as suggested by Rohl [Roh75], [BiB80];

c. local display implementation as in ICL 2900 Pascal compiler [Ree80] and also chosen for a virtual architecture for Ada [Dom80];

d. implementation of Tanenbaum's proposal [Tan78].

When the architect decides to support some of the HLL functions through an exo-architectural support [Das82a], the set of such supports for a particular function(s) will be referred to as an exo-architectural component or simply an architectural component. For example, the set of the architectural supports for the HLL functions of procedure Call & Return could be viewed as an architectural component.

An examination of the execution of such architectural components on existing Von-Neumann style of architectures reveal that the transfer operation is by far the most predominant operation. The transfers could be between (a) two memory locations, (b) a processor register and a memory location or (c) two processor registers. So it is quite reasonable to compare the implementation techniques in terms of number of transfer operations required to implement an architectural component on a target processor. If we consider appropriate weights for the different kinds of transfer and the necessary statistics of program behaviour are known, the average cost of execution of a particular implementation of an architectural component (in terms of transfer weights) on a host processor could be obtained. This is essentially a technique for analysis of suitability of host processor support for implementing some HLL functions. A somewhat similar study for two of the above mentioned implementation techniques (a & d) was done by DePrycker [Dep82b]. But the method proposed here should be essentially considered as a design aid rather than a technique for analysis (though it could very well be used for that purpose in subsequent phases of the design process). To start with, there is no need to assume any underlying hardware/firmware organization of the processor excepting in that typical Von-Neumann style of processor design [Das83a] will be adopted.

In the absence of any assumptions regarding the underlying hardware/firmware structure, it is necessary to use an abstract way of describing a particular technique for implementation of an architectural component. The computer design & description language (CDDL) S*A [Das81, Das82a, Das82b] is used for that purpose. From an S*A description of an architectural component, two measures could be derived that are termed as Virtual Transfer Complexity (VTC) and the average VTC(AVTC). As explained in Section 3.2, the VTC measure is only a design aid in the sense that it indicates the minimum transfer complexity that could be theoretically achieved using a particular technique of implementation. One of the important characteristics of an S*A 'mechanism' is that the description of the mechanism remains independent of any change in the underlying support. The same is true for the AVTC measure except in that it is dependent on the usage statistics obtained from the programming language environment under consideration.

In the next section, after presenting an overview of how the descriptions of the architectural components using the CDDL S*A facilitate the design process, the notion of Virtual Transfer Complexity will be developed and some illustrative evaluation of VTC for a few typical S*A statements will be included.

Section 3.2 includes a brief review of some of the relevant notions of maintenance and variable access in the run time representation of a tyical block structured HLL environment as that of Ada. Complete S*A descriptions of the procedure Call-Return (CR) and Block Entry-Exit (BE) components for all the four implementation techniques are presented in this section. The section also includes the evaluation of AVTC's of the components for each technique.

The remaining sections are devoted to deriving the necessary processor support from the S*A descriptions and evaluating the real transfer complexity (RTC) of the components on this organization. A comparison of the AVTC and ARTC values indicate that the choice of the implementation techniques could be restricted to only the techniques suggested by Rohl and Tanenbaum. Moreover, it is concluded that the choice between these two techniques depends on the relative frequency of procedures being declared at an intermediate (neither local nor global) lexical level with respect to the calling level.

3.1 S*A Description and Virtual Transfer Complexity

The S*A description facilitates the design process as follows:

a. It is possible to describe the functioning of an architectural component at a level of abstraction that makes the description invariant with respect to the underlying hardware/firmware supporting the implementation of the component.

b. It is possible to formally verify the correctness of a mechanism before any further refinement of the abstraction in the design process [Das83b].

c. The description clearly indicates the dependence of the architectural component on specific statistics of the programming language environment. This feature facilitates collection of appropriate statistics of usage.

d. The description clearly indicates the possible trade-offs between M & R measures [DiS79] necessary for performance improvement. If all the important architectural components of a language directed architecture are considered together, the S*A descriptions of the components could yield a fairly complete hardware/firmware requirements for the processor.

The Virtual Transfer Complexity (VTC) of the S*A description of an architectural component, implemented through a particular technique is a measure of the cumulative count of number of transfers necessary for the execution of the component. While evaluating the count, all the local variables (Privars in S*A) and the global variables (Glovars in S*A) are assumed to be available in the Processor registers of a virtual Processor. The array types are considered as register banks with usual selection mechanism and a transfer through ALU is considered as a register-register transfer between ALU input to ALU output. Presently any parallel execution of S*A statements will not be considered, yet any such parallelism could be easily accounted for by only considering the statements having a maximum transfer count out of a few parallel statements.

In the next section VTC and AVTC of the procedure Call-Return component and the Block Entry-Exit component for all the four implementation techniques (mentioned earlier) would be evaluated. Before proceeding to the next section, the method of evaluation of VTC of some typical S*A statements is explained. The following type declarations are assumed:

```
type   M = Seq [..] bit ;
type   A, B = array [..] of M ;
       X, Y, P, Q : M;
```

a. X := P - Q;

(An abstract Arithmetic unit is assumed with two inputs ALU-1 & ALU-2 and the output ALU-0).

The transfers are: from P to ALU-1; from Q to ALU-2; a transfer through ALU and a transfer from ALU-0 to X. Thus the VTC of this statement is 4.

[Note: It is reasonable to assume the presence of an ALU with 2 inputs and 1 output capable of performing standard arithmetic and logical operations used in the S*A descriptions of the components. In evaluating the VTC for 'increment/decrement by 1' operations, we distinguish it from standard Add & Subtract operations in the sense that only one operand is considered to be transferred to the input.]

b. P := A[X];

The transfers are: transfer of X to the selection mechanism of the array A; transfer of the selected array component to P. The VTC of the statement is 2.

c. A[X] := B[P + A[Q]];

The transfers are: transfer of Q to selection mechanism of A; transfer of the selected component of A to ALU-1; transfer of P to ALU-2; transfer through the ALU; ALU-0 to selection mechanism of B; transfer of X to selection mechanism of A; transfer of selected component in B to the selected component in A. The VTC of the statement is 7.

d. WHILE P ≠ Q DO....OD;

If cumulative VTC of the statements between the delimiters DO and OD is w and if the DO-loop is executed m times, the VTC of the above compound statement is evaluated to be mw+3(m+1). The transfer involved in the branching operation is ignored.

3.2 Variable Addressing Techniques

A program in a block-structured language may be depicted as a tree to represent the nesting of the blocks [Dor79]. A block is delimited by **begin** and **end**. Thus each block may be associated with a level in the tree and is called the static lexical level. The execution of a program may be viewed as dynamic changing of lexical levels. The lexical level of execution is called the dynamic lexical level.

From a static point of view (i.e., at compile time) a block and a procedure (delimited by **Procedure** and **end**) are treated identically. The body of the procedure (delimited by **begin-end** pair) is treated naturally as a block and is associated with a lexical level one higher than the level of declaration of the corresponding procedure identifier. Within a block a variable is associated with a (lexical level, sequence number) pair. The lexical level component of the pair corresponds to the static lexical level of the block where the variable is declared, the sequence number indicates the sequence of declarations in the block.

The dynamic change in lexical level occurs due to a block invocation or a procedure call. The nature of change in each case could be quite different. The new dynamic level due to a block invocation corresponds to the static level of the invoked block in the static program structure. A procedure invocation changes the dynamic level to correspond to a level one higher than the static level of declaration of the procedure identifier. The exit out of a block or return from a procedure requires the dynamic level to be reset to the level of invoking the block or calling of the procedure.

In a block-structured language, a variable may be accessed if it is declared in the same block or in a statically surrounding block, i.e., when the variable is declared in one of the levels corresponding to the nodes in the path from the root to the node at the level of access in the tree. So the set of nodes on the path from the root to the active node could be considered to be characterizing the lexical levels involved in the dynamic variable accessing environment. In case of procedure calls, the procedure identifier is treated as a variable asociated with a (lexical level, sequence number) pair and so the rules for procedure invocability is the same as that of accessibility of a variable.

In the Ada environment, blocks could be treated as degenerate procedures called from the level where they are defined. The above discussion indicates that two linked list structures (or stacks) are needed to appropriately represent the static and dynamic structures of the environment. Usually the evaluation/allocation stack is combined with these two structures to form an activation stack. The stack frames corresponding to the static program structure are linked through a static link and the history of dynamic program activity is maintained through a dynamic link in each stack frame. It might be noted that the dynamic and static links in a frame allocated due to a block invocation contain the same values and merely point to the previous frame. In a frame allocated to the body of a procedure, the dynamic link points to the previous frame but the static link points to the frame where the procedure identifier was allocated. The dynamic link information is used in reverting back to the dynamic calling or block-invocation level after a return from a procedure or exit from a block. The static link information is utilised in accessing non-local variables.

With the above description in mind, accessing a variable in the parent static environment requires following the static link chain to the appropriate level of declaration. Thus depending on the lexical level difference of access and declaration, a variable access could involve quite a few levels of indirection. Dijkstra proposed the technique of using an extra set of display locations (as a stack) to reduce this overhead [Dij60, Bac79, Hor83]. In a variable accessing mechanism implemented using Dijkstra's proposal, any accessible lexical level could be directly reached through the corresponding display location associated with that level. The top of the display stack points to the presently active frame and the remaining display contains the copy of the static links of the accessing environment. The exact description of the sequences involved in procedure Call-Return and Block Entry-Exit are described in the S*A description of the mechanism M_1. For ease of understanding the description, Fig 3.1(a) depicts the base of a stack frame used by this technique. A major problem with this classical technique is in rebuilding the display bank on return from a procedure; the number of locations to be reset is directly proportional to the lexical level difference between the levels of calling and declaration of the procedure.

A modification to the above technique was suggested by Rohl [Roh75]. He observed that there was duplication of information among the static links, dynamic links and the display. The sequences involved in procedure call-return and block entryexit for the suggested modification is given in the S*A description of the mechanism M_2. In this scheme the static and dynamic links are replaced by a single link that links only the frames that are associated with the same lexical level. This implies that whenever the content of a display location is overwritten (for a call or entry) the corresponding information is stored in the stack frame associated with that dynamic lexical level. So it could be observed that the display rebuilding overhead after a return from a procedure is independent of the lexical level difference between the calling level and the level of declaration of the procedure. The rebuilding involves only the resetting of a single display location associated with the static lexical level of the procedure body. Figure 3.1(b) depicts the base of a stack frame when this technique is used.

Another variation of a technique that implements variable addressing through display locations is generally known as the local display method [BiB80, Dom80]. The procedures for procedure call-return and block entry-exit using this technique are described in the S*A mechanism M_3. In this method the display locations necessary to represent the current variable accessing environment are stored in the current frame of the activation stack. The current frame is deallocated on return from Procedure or exit from a block. Thus there is practically no extra overhead for rebuilding the display on return from a procedure.

A significant variation from the above techniques was suggested by Tanenbaum [Tan78]. His observation was that most of the variable accesses are local or global in nature (not exactly true for an Algol-like language) and thus dedicated display locations for non-local references were not necessary. He proposed the use of two dedicated locations or registers for pointing to the bases of the current frame and global frame of the activation stack. The procedures for procedure call-return and block entry-exit are described in the S*A mechanism M_4. In a previous analysis, DePrycker [Dep82a] ignored the overhead involved in accessing the procedure identifier during a procedure Call, when the procedure identifier is neither a local nor a global variable. In Tanenbaum's description this overhead is included in the precall sequence (Mark instruction) [Tan78]. In this technique, any variable that is neither local nor global is accessed using the chain of static links.

Note: [In the S*A mechanisms to follow, lexical level of the root node of the program tree is considered to be 1. Moreover, a variable is associated with a (lexical level, offset) pair, where offset is with respect to the base of allocation/evaluation area in a static frame].

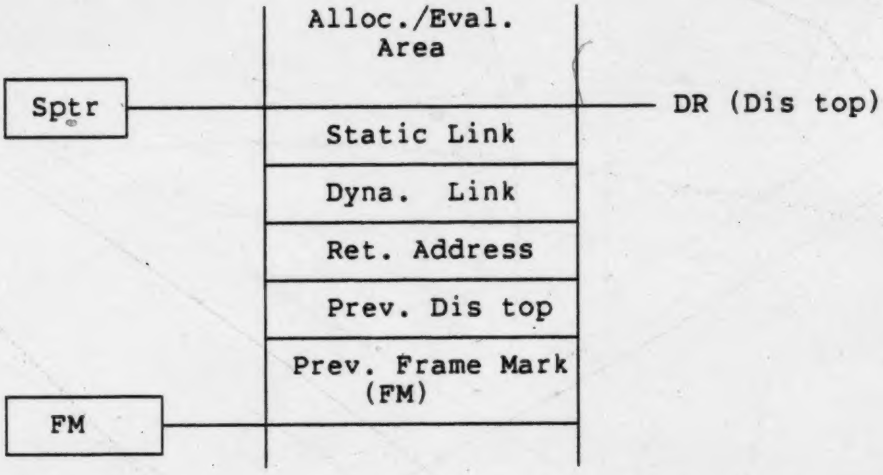

Alloc./Eval. Area
Sptr
DR (Dis top)
Static Link
Dyna. Link
Ret. Address
Prev. Dis top
Prev. Frame Mark (FM)
FM

Figure 3.1(a)

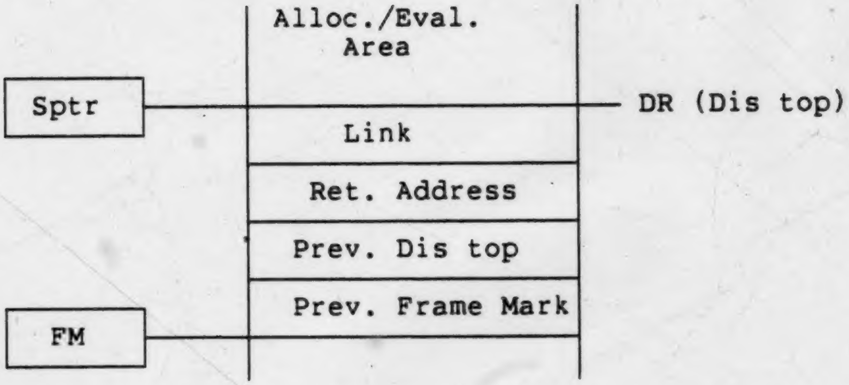

Alloc./Eval. Area
Sptr
DR (Dis top)
Link
Ret. Address
Prev. Dis top
Prev. Frame Mark
FM

Figure 3.1(b)

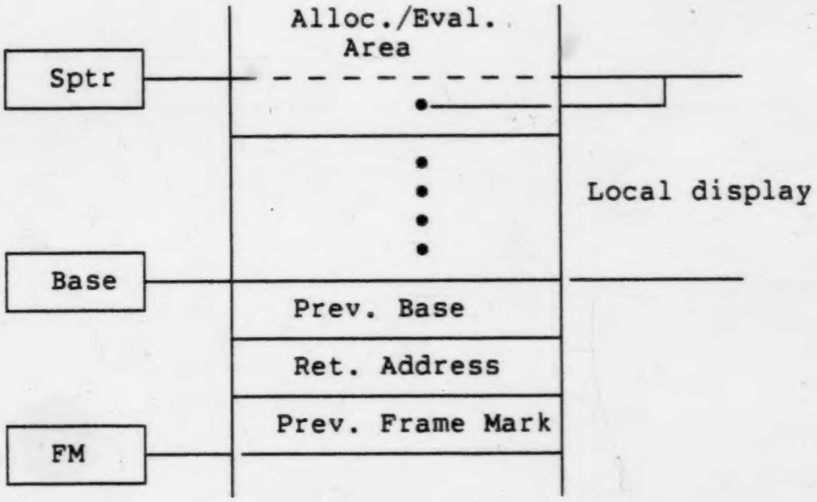

Alloc./Eval.
Area
Sptr
Local display
Base
Prev. Base
Ret. Address
Prev. Frame Mark
FM

Figure 3.1(c)

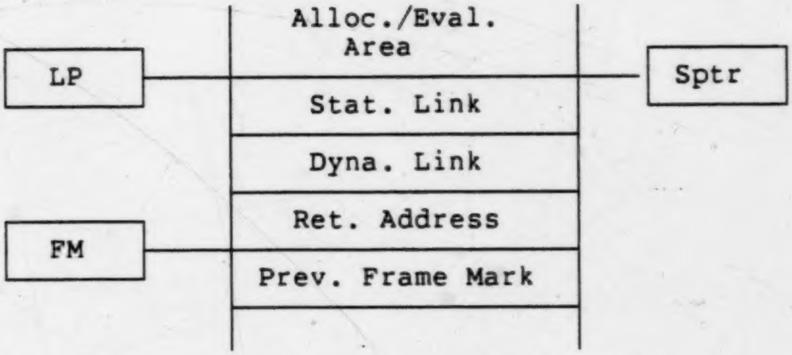

Alloc./Eval.
Area
LP
Sptr
Stat. Link
Dyna. Link
Ret. Address
FM
Prev. Frame Mark

Figure 3.1(d)

Figure 3.1 : A View of the Current Stack Frame Just After Execution of the Procedures (a) M_1.CALL (b) M_2.CALL (c) M_3.CALL (d) M_4.CALL

```
Mech $M_1$ ; /* DIJKSTRA's DISPLAY MECHANISM */
     type memword = seq [..] bit;
     glovar mainmem: array [..] of memword;
     syn stack = mainmem;
     glovar Sptr, Frame-mark, Pctr : memword;
     glovar Display : array [0..$2^{b+1}$ -1] of memword;
     glovar Inst-reg : tuple
                       adr : tuple
                            Lex: seq[..] bit;
                            Offset : seq [..] bit;
                            endtup
                       endtup;
                       :Memword;
     privar local, stat, ptr : memword;
     privar Dis-top : seq [b..0] bit

Proc CALL;
     Stack [Sptr] := Frame-mark ; (2)
     Frame-mark := Sptr ;(1)  Sptr:=Sptr + 1 ; (3)
     Stack [Sptr] := Dis-top ; (2) sptr := sptr + 1 ; (3)
     /* stored the 'calling' lexical level number */
     Stack [sptr] := Pctr ; (2) Sptr := Sptr  + 1 ; (3)
     /* Stored the return address */
     Stack [Sptr] := Display [Dis-top] ; (3)
     sptr := sptr + 1 ; (3)
     /* Stored the dynamic link */
     Dis-top := Inst-reg. adr. Lex (1)
```

```
      /* Dis-top gets the lexical level of the block where
           the procedure identifier was declared */
      Stack [Sptr] := Display [Dis-top] ; (3)
      sptr := sptr + 1 ; (3)
      /* stored the static link */
      Pctr := Stack [Display[Dis-top+Inst-reg.adr.offset]];
      /* Address of the code segment of the called
         routine is loaded in Pctr */
      Dis-top := Dis-top + 1 ; (3)
      /* Dis top contains lexical level of the declaration
          of the procedure body */
      Display [Dis-top] := Sptr ; (2)
endProc;
Proc RETURN ;

      Sptr := Frame-mark ; (1)
              Frame-mark := Stack [Sptr] ; (2)
      Local := Display [Dis-top] ; (2)
      Dis-top := Stack [Local - 4] ; (5)

             /* Calling lex level in Dis-top */
      Pctr := Stack [Local - 3] ; (5)
               /* Return address in Pctr */
      Display [Dis-top] := Stack [Local - 2] (6)
      /* display [Dis-top] points to calling environment */
      Stat := Stack [Local - 1] ; (4)
                /* static link in stat */
      Ptr := Stack [Display [Dis-top] - 1] (5)
      /* Stat and Ptr are used to facilitate rebuilding
          of Display */
      Local := Dis-top ; (1) Local := Local - 1 (3)
      /* Dis-top is kept undisturbed */
      WHILE Display [Local]¬ = stat (3[n+1])
      DO    Display [Local] := Ptr ; (2)
            Ptr := Stack [Ptr-1] ; (4)
      /* Ptr is used in traversing the static chain */
               Local := Local - 1 ; (3)
      OD ;  /* (n+1) = lex level diff. between
       the calling level and level where
       procedure identifier was declared */
endProc;
Proc ENTRY;
      STACK [Sptr] := Frame-mark ; (2)
            Frame-mark := Sptr; (1)  Sptr := Sptr + 1 ; (3)
      Stack [Sptr] := Display [Dis-Top] ; (3)
                                Sptr := Sptr + 1 ; (3)
      /* Stored dynamic link */
      Stack [Sptr] := Display [Dis-Top] ; (3)
                                 Sptr := Sptr + 1 ; (3)
      /* stored static link; though it is unnecessary. */
      /* for the sake of uniformity */
      Dis-top := Dis-top + 1 ; (3)
      Display [Dis-top] := Sptr ; (2)
endProc;
```

```
Proc  EXIT;
      Sptr := Frame-mark ; (1)
              Frame-mark := Stack [sptr] ; (2)
      Dis-top := Dis-top + 1 ; (3)

endProc

endMech $M_1$;
```

```
Mech $M_2$ ; /* ROHL's METHOD */

      type memword = Seq [..] bit;
      glovar mainmem : array [..] of memword;
      syn Stack = mainmem;
      glovar Sptr, Frame-mark, Pctr : memword;
      glovar Display : array [0..$2^{b+1}$ -1] of memword;
      glovar Inst-reg : tuple
                            adr : tuple
                                  Lex: Seq[..] bit;
                                  Offset : [..] bit;
                                  endtup
                            endtup;
                            : memword;
      privar Dis-top : Seq [b..0] bit
      privar Local : memword

Proc CALL;
      Stack [Sptr] := Frame-mark ; (2)
             Frame-mark := Sptr ; (1)  sptr:= sptr + 1 ; (3)
      Stack [sptr] := Dis-top ; (2) Sptr := Sptr + 1 ; (3)
      /* lex level number of calling level stored */
      Stack [sptr] := Pctr ; (2) sptr := Sptr + 1 (3)
      /* ret. addr. stored */
      Dis-top := Inst-reg. adr. lex ; (1)
      Pctr := Stack [Display [Dis-top]
```

```
                + Inst-reg.Adr. Offset] (6)
        /* Pctr is loaded with addr. of code segment of the
        called procedure */
        Sptr := Sptr + 1 ; (3)
        Dis-top := Dis-top + 1 ; (3) /* Dis-top points to the
        head of present display;  Lex level of the block for
        procedure body is 1 more than the block where Proc.
        identifier was declared */
        Stack [Sptr] := Display [Dis-top] ; (3)
                                sptr := sptr + 1 ; (3)
        /* Content of Display location is stored as it will be
        overwritten with a pointer to  new level */
        Display [Dis-top] := Sptr ; (2)
endProc;

Proc RETURN ;
      Sptr = Frame-mark ; (1) Frame-mark := Stack [sptr] ; (2)
      /* Deallocation of frame */
      Local := Display [Dis-top] ; (2)
      Display [Dis-top] := Stack [Local-1] (5)
      /* reinstated the content that was destroyed due
            to call */
      Dis-top := Stack [Local - 3] ; (5)
      Pctr := Stack [Local - 2] ; (5)
EndProc ;

Proc ENTRY ;
```

```
      Stack [Sptr] := Frame-mark ; (2)
      Frame-mark := Sptr; (1) Sptr := Sptr + 1 ; (3)
      Dis-top := Dis-top + 1 ; (3)
      Stack [Sptr] := Display [Dis-Top] ; Sptr := Sptr + 1 ;
      /* As in call */

      Display [Dis-top] := Sptr ; (3)
      /* Display [Dis-top] points to present environment */
endProc ;

Proc  EXIT ;
      Sptr := Frame-mark ; (1)
              Frame-mark := stack [sptr] ; (2)
      Display [Dis-top] :=
           Stack [Display [Dis-top] - 1] ; (6)
      /* restored the content overwritten due to
         Block entry */
      Dis-top := Dis-top - 1 ; (3)
EndProc ;

End Mech M2 ;
```

```
Mech M3 ; /* LOCAL - DISPLAY */

      type memword = seq [..] bit;
      glovar mainmem: array [..] of memword;
      syn stack = mainmem;
      glovar Sptr, Frame-mark, Pctr : memword;
      glovar Inst-reg : tuple
                             adr : tuple
                                   Lex: Seq[..] bit;
                                   Offset : seq [..] bit;
                                   endtup
                             endtup;
                             : memword;
      glovar Base : memword;
      privar Ptr : memword ;

Proc CALL;
      Stack [Sptr] := Frame-mark ; (2)
      Frame-mark := Sptr ; (1)  Sptr:= Sptr + 1 ; (3)
      Stack [Sptr] := Pctr ; (2) Sptr := Sptr + 1 ; (3)
      /* return address stored */
      Stack [Sptr] := Base ; (2) Sptr := Sptr + 1 (3)
      /* dynamic link stored */
      Ptr := Base ; (1) /* copy of dyn. link in Ptr */
      Base := Sptr ; (1) /* new base assigned */
      /* The following loop builds up the necessary local
```

```
        display in the present frame */
        WHILE Inst-reg. adr. Lex ¬= 0 (3[ β + 1])
             DO
                 Stack [Sptr] := Stack [ptr] ; (3)
                 Sptr := Sptr + 1 (3)  Ptr := Ptr + 1] ; (3)
             Inst-reg. adr. lex := Inst-reg. ad. lex - 1 (3)
             OD ;  /* β = lex level of called
                                   procedure identifier */
        Stack [Sptr] := Sptr + 1 ; (4)
             /* topmost display stored */
                        Sptr := Sptr + 1 ; (3)
        Pctr := Stack [Stack [Base + Tlex]
                + Inst-reg.adr. Offset]; (7)
        /* Pctr is loaded with addr. of the code segment of
        the called pgm. */
  EndProc;

  Proc RETURN ;

        Sptr := Frame-mark ; (1)
                Frame-mark := Stack [Sptr] ; (2)
        Pctr := Stack [Base - 2] ; (5) /* ret. adr. loaded */
        Base := Stack [Base - 1] ; (4)
        /* Base points to base of Calling Frame */

  EndProc ;
```

```
Proc ENTRY ;
      STACK [Sptr] := Frame-mark ; (2)
                              Frame-mark := sptr; (1)
                      Sptr := Sptr + 1 ; (3)
      /* new frame allocated */
      Stack [Sptr] := Base ; (2) Sptr := Sptr + 1 ; (3)
      Ptr := Base (1)
      Base := Sptr ; (1) /* new base */
WHILE     Stack [ptr] - 1 ¬ = Ptr (6[ν + 1])
   DO     Stack [Sptr] := Stack [ptr] ; (3)
          sptr := Sptr + 1 ; (3)
          ptr := Ptr + 1 ; (3)
   OD;
      Stack [Sptr] := + 1 ; (4) /* As in Proc. CALL */
      Sptr := sptr + 1 ; (3)
EndProc ;

Proc EXIT  ;
      Sptr := Frame-mark ; (1)
              Frame-mark := Stack [sptr] ; (2)
      /* De-allocation of frame */
      Base := stack [Base - 1] ; (4)
endProc ;

end Mech M₃ ;
```

```
Mech M4 ; /* TANENBAUM's METHOD */
      type memword = seq [..] bit;
      glovar mainmem: array [..] of memword;
      syn stack = mainmem;
      glovar sptr, Frame-mark, Pctr : memword;
      glovar LP, GP : memword;
      glovar Inst-reg : tuple
                            adr : tuple
                                  Dlex: seq[..] bit;
                                  Offset : seq [..] bit;
                                  endtup
                            endtup;
                            : memword;
      privar chain, Ptr : memword ;

Proc CALL;
      Stack [Sptr] := Frame-mark ; (2)
      Frame-mark := Sptr ; (1)  Sptr:=Sptr + 1 ; (3)
      Stack [Sptr] := Pctr ; (2) Sptr := Sptr  + 1 ; (3)
      Stack [Sptr] := LP ; (2) sptr := Sptr + 1 ; (3)
      /* Dynamic link stored */
      If Inst-reg. Adr. Dlex = 'Global' (3)
            Stack [Sptr] := GP ; (2)  Ptr := GP; (1)
      ||Inst-reg. Adr. Dlex = 0 (3) Stack [sptr] := LP ; (2)
                                              Ptr := LP; (1)
      ||DO chain := Inst-reg.Adr. Dlex ; (1)
```

```
            WHILE Chain ¬= 0 (3 [m + 1])
            DO    Ptr := Stack [Ptr - 1] ; (4)
                        Chain := Chain - 1 ; (3)
            OD

              Stack [Sptr] := ptr ; (2) Sptr := Sptr + 1 ; (3)
              /* static link set */
            OD  /* m = lexical level difference between
              the level of call and the level of declaration
              of the procedure identifier , when the procedure
              identifier is an intermediate variable . */
          fi ;
            LP := Sptr ; (1) /* New LP set */
            Pctr := Stack [ptr + Inst.reg. adr. offset] ; (5)
    EndProc ;

    Proc RETURN ;
          Sptr := Frame-mark ; (1)
                  Frame-mark := stack [sptr] ; (2)
          Pctr := Stack [LP - 3] ; (5) /* ret. address  */
          LP : Stack [LP - 2] ; (5) /* LP points to
                Calling environment */
    endProc;
    Proc ENTRY ;
          Stack [sptr] := Frame-mark ; (2)
          Frame-mark := Sptr; (1)  Sptr := Sptr + 1 ; (3)
          Stack [sptr] := LP ; (2) sptr := sptr + 1 ; (3)
```

```
      Stack [sptr] := LP ; (2) sptr := sptr + 1 ; (3)
      /* both links are stored, for uniformity in var
         access mem */
      LP := Sptr ; (1)
endProc ;

Proc  EXIT ;
      Sptr := Frame-mark ; (1)
              Frame-mark := Stack [Sptr] ; (2)
      LP := Stack [LP - 1] ; (4)

endProc

end Mech $M_4$ ;
```

The virtual transfer complexity of CR and BE components of the above-mentioned mechanisms are now evaluated .

1 a) VTC of M_1.CR (Call-Return component as described in Mech M_1)

$= 39 + 34 + 9n + 3\,(n + 1)$

$= 64 + 12(n + 1) = \underline{64} \ \underline{+}\ \underline{12}\ \delta_k$;

where δ_k represents the difference in lexical level between the point of calling and point of declaration of the Procedure identifier.

The AVTC of M_1.CR $= 64 + 12\hat{\delta}_k$, where $\hat{\delta}_k$ is the average δ_k computed from a large set of sample programs characterizing the programming language environment under consideration.

1 b) VTC of M_1.BE (Block Entry-Exit Component as described in mech M_1)

$= 23 + 6 = \underline{29}$ = AVTC of M_1.BE

2 a) VTC of M_2.CR (Call-Return component as described in Mech M_2)

$= 37 + 20 = \underline{57}$ = AVTC of M_2.CR

2 b) VTC of M_2.BE (Block Entry-Exit component as described in Mech M_2)

$= 12 + 12 = \underline{24}$ = AVTC of M_2.BE.

3 a) VTC of M_3.CR

$= 35 + 12\beta + 3(\beta + 1) + 12 = \underline{50} \underline{+} \underline{15} \underline{\beta}$;

where β is the lexical level of the procedure identifier.

AVTC of M_3.CR = $50 + 15\beta$;

where $\hat{\beta}$ is the average β computed over a large number of sample programs.

3 b) VTC of M_3.BE

$= 20 + 9\gamma + 6(\gamma + 1) + 7 = \underline{33} \underline{+} \underline{15\gamma}$ [where γ = number of displays to be transferred from previous frame] and AVTC of M_3.BE = $33 + 15\hat{\gamma}$.

4 a) VTC of M_4.CR =

$(22 + 6) + 13 = 41$ [if procedure identifier is a global variable] or

$(22 + 3 + 5) + 13 = 44$ [if procedure identifier is a local variable at calling level] or

$(22 + 3 + 3 + (10\,\delta'_k + 13)) + 13 = 54 + 10\,\delta'_k$ [otherwise].

AVTC of M_4.CR =

$35 + \eta_x(6) + \eta_t(9) + \eta_y(29 + 10\,\hat{\delta}'_k)$;

where η_x = relative frequency of a procedure identifier being a global variable.

η_t = rel. freq. of a procedure identifier being a local variable.

η_y = relative frequency of a procedure identifier being an intermediate variable.

$\hat{\delta}'_k$ = mean difference in lexical level between the level of calling and declaration for a procedure identifier that is declared at an intermediate level. $\hat{\delta}'_k$ is approximately considered equal to $\hat{\delta}_k$ as $\hat{\beta} \approx 1$.

The relative frequencies are assumed to be obtained by considering all the procedure calls (static) over a large number of sample programs characterizing the programming language environment.

4 b) VTC of M_4.BE = 24 = AVTC.

3.2.1 Comparison of the AVTCs

For the purpose of comparison, the following preliminary statistics obtained by DePrycker [Dep82b], for Algol-60, are used : $\hat{\delta}_k \simeq 2$, $\hat{\beta} \simeq 1$ and $\hat{\nu} \simeq 2$.

Using these values in the comparison of M_1.CR, M_2.CR and M_3.CR , it could be observed that

AVTC (M_1.CR) $\cong$88 ; AVTC (M_2.CR) = 57 ; AVTC (M_3.CR) $\cong$ 65.

As far as the CR components of these three mechanisms are concerned, M_2.CR is expected to perform best. Similarly, comparing the AVTC's of the BE components of these three mechanisms, it is observed that

AVTC (M_1.BE) = 29 ; AVTC (M_2.BE) = 24 and AVTC (M_3.BE) $\cong$ 63.

Again the conclusion is that the M_2.BE is expected to perform best. It is difficult to make an absolute comparison between M_4.CR and the CR components of the other three mechanisms, as the program statistics η_t, η_x, η_y are not available in the literature.

Still it is possible to make an interesting observation when AVTC (M_2.CR) and AVTC (M_4.CR) are compared. The CR and BE components of M_4 with that of M_1 and M_3 are not compared, as it has already been observed that M_2 is definitely superior in all respects. The AVTC measures of (M_2.BE) and M_4.BE) are equal.

Let $(\eta_t + \eta_x) = \eta_z$

then, $\text{AVTC}(M_4.\text{CR}) \cong 35 + \eta_z.(7.5) + \eta_y.(29 + 20)$

$$= 35 + 7.5\ \eta_z + 49\ \eta_y$$

We know, AVTC (M_2.CR) = 57 and $\eta_z + \eta_y = 1$.

From the above three relations, it is possible to conclude that if the relative frequency of declaring a procedure at an intermediate level is greater than 0.4 then AVTC (M_4.CR + M_4.BE) > AVTC (M_2.CR + M_2.BE)

For choosing the implementation technique for the variable address mechanism in a block-structured environment, it is necessary to include the VTC of variable access. From the description of the four mechanisms, it is obvious that the VTC of accessing a variable is same for M_1 and M_2 and it is less than the VTC of a variable access in M_3. The discussion on the average cost of using any one of these implementation techniques is postponed until a later section, where the cost of variable access is also included in the evaluation. In that section some interesting observations regarding the choice among the mechanisms are restricted to the mechanisms M_2 and M_4, as the AVTC's (as well as the ARTC's, in the next section) of the CR and BE components of M_1 and M_3 have been found to be larger.

3.3 The Real Transfer Complexity

Given the AVTC's, the designer knows the best that could be derived out of the implementation techniques. Now the designer is faced with the problem of choosing a technique for implementation on a real architecture keeping the various design constraints and economic considerations in mind. The choice based on AVTC's might not be the same on a real hardware/firmware combination unless an 'ideal' range for mapping the domain of S*A constructs onto the hardware/firmware combination is available. (The following discussion assumes a microprogram controlled processor, though the conclusions obtained would be valid for an equivalent hardwired control scheme.) As indicated earlier, the S*A description of a mechanism clearly suggests the hardware/firmware support necessary for a most effective implementation. Once the design constraints are clearly laid out the architect can easily specify the feasible hardware/firmware support of the processor. The development of a semi-automated design support through a family of description languages is in progress [DaO82].

The real transfer complexity of an S*A statement is expressed as a matrix of three components namely, the number of register-register transfers ($R(S_1)$), number of transfers through the ALU ($A(S_1)$) and the number of transfers across the processor-memory interface ($MP(S_1)$) required for the execution of the statement S_i in the S*A descrition. So $RTC(S_i) = [(R(S_i))_R \ (A(S_i))_A \ (MP(S_i))_M]$, the subscripts R, A and M are used for no other reason but notational clarity. The RTC of an exo-architectural component X in mech M_i is denoted as RTC $(M_i.X)$ and is $\sum_i RTC(S_i)$, the summation includes all the statements in the execution of the component $(M_i.X)$. The average RTC (ARTC) is calculated as before.

In the S*A description of the architectural components, introduction of any possible parallelisms among various statements is intentionally avoided . Similarly it is assumed that the microprogram control of the real processor is of purely vertical nature. The simplistic assumptions are maintained so that the primary issues in the methodology are clearly laid out. It might be noted that the language S*A has adequate constructs to express parallelism [Das82a] and once the mechanisms include parallelism, the underlying firmware control (to be designed) could include possible parallel operations. The evaluation of VTC and RTC requires quite simple modifications.

Before evaluating the RTC's of the components in discussion, it is necessary to propose the organization of the hardware/firmware combination of the processor that would support these components. As indicated earlier, the requirements of the supports necessary to achieve a RTC measure reasonably close to the VTC measure could be directly obtained from the S*A descriptions.

Instead of describing four different processor organizations for efficiently supporting the implmentation of the components of the four mechanisms, Fuller's Canonical Processor is adopted as the basic Von-Neumann structure and any special requirements for each mechanism are indicated. The processor organization is shown in fig. 3.2. It is assumed that any two of the registers (shown in the figure) could participate in a register-register transfer through a common bus.

The stack is assumed to be in the main memory. The Glovar Display is mapped on to a bank of registers(DR's) and the privar Dis-top is mapped on to a register Dis-top which is used as a selector index for the bank of registers DR. The glovars Sptr, Pctr and Frame-mark are mapped onto the processor registers SP, PC (Program Counter) and FM ; Inst-reg is mapped onto IR (Instruction Register). The privars LOCAL (in M_1 and M_2) as well as the Glovar LP (in M_4) are mapped onto the processor register L. The Privar Ptr is mapped onto the register PT. The Privars stat (in M_1), Base (in M_3) and the Glovar GP is mapped onto the processor register G. The Privar Chain is mapped onto the register T. At this point no distinction is made between the registers that are user-addressable and the registers that are only used by the microprograms. Such a specification is quite straightforward if the analysis/design is restricted to a specific mechanism. In any case the microcode can address any one of the registers.

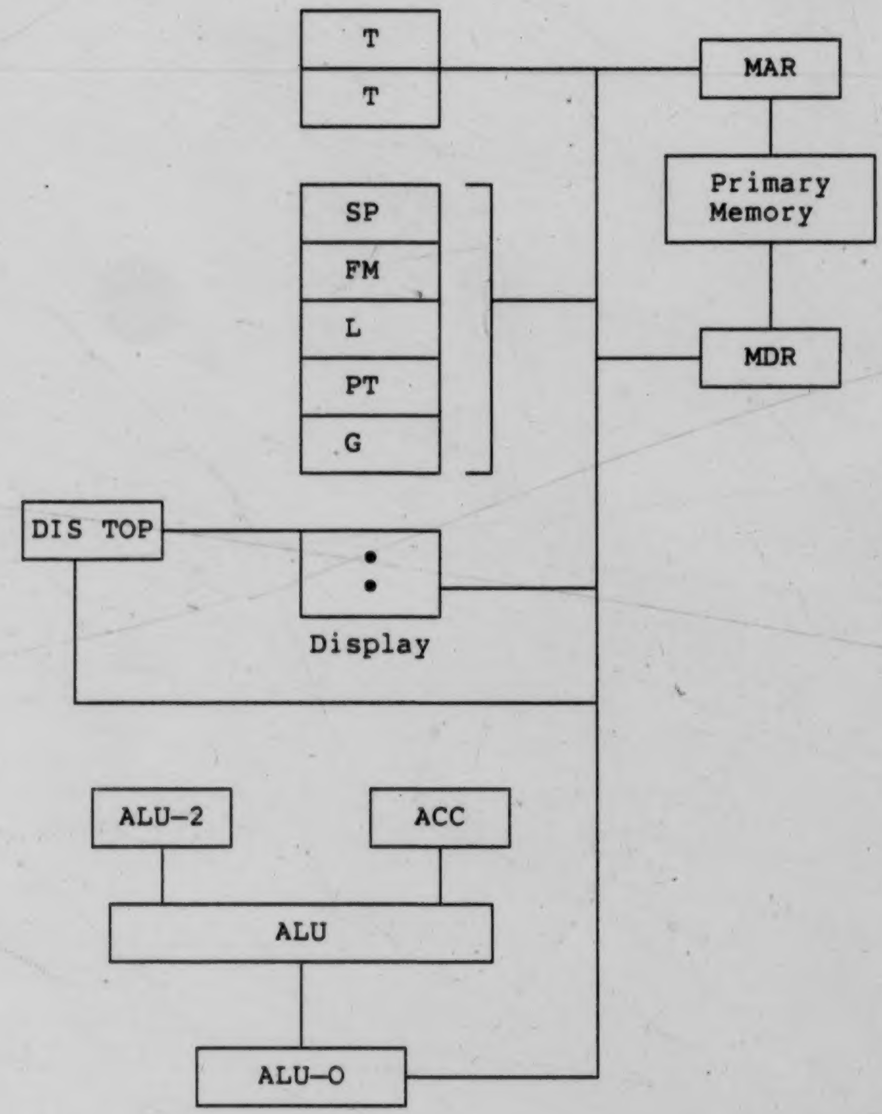

T
T
MAR
Primary Memory
SP
FM
L
PT
G
MDR
DIS TOP
Display
ALU-2
ACC
ALU
ALU-O

Figure 3.2 The Processor Organization.
(control lines are not shown)

The temporary register T_2 is to be only used by the microcode. The register ACC serves as one of the inputs to the ALU. ALU-2 and ALU-0 are respectively an input and the output registers of the ALU. For any monadic ALU operation, the operand is expected in the ACC.

The choice of this particular processor structure is a direct implementation of the S*A description on a typical Von-Neumann structure with a purely vertical microprogram control where obvious design constraints lead to the implementation of the Glovar stack in main memory with a constant address base (St-base) known to the firmware. It could be noted that the processor structure is not biased towards any particular mechanism and thus the comparison of RTC's based on this processor structure could be considered fair.

Examples of RTC evaluation for some typical S*A statements involved in describing the mechanisms are presented below.

1. **Stack [sptr] := Frame-mark;**

   Frame-mark and Sptr are available in processor registers. The stack is in main memory and the stack-base is a constant address known to the firmware or the underlying hardware control. Only the transfers involved are indicated and the transfers are tagged with R, M or A to indicate the nature of the transfer. The transfers are:

   Stack-base ----> ALU-2 ; (R)

SP ---> ACC ;(R) ADD ;(A) ALU-0 ---> MAR ;(R)

Frame-mark ---> MDR ;(R) WRITE ;(M)

The RTC of the statement is $[4_R\ 1_A\ 1_M]$

2. **Stack [Sptr] := Display [Dis-top] ;**

   As mentioned earlier Display is a bank of registers in the processor indexed by the register Dis-top. The transfers are:

   Display [Dis-top] ---> MDR ; 2(R)

   Stack-base ---> ALU-2 ;(R) SP ---> ACC ;(R)

   ADD ;(A) ALU-0 ---> MAR ;(R) WRITE ;(M)

   The RTC of the statement is $[5_R\ 1_A\ 1_M]$

3. **Sptr := Sptr + 1 ;**

   The transfers are:

   SP ---> ACC ;(R) INCREMENT ;(A) ALU-0 ---> SP ;(R)

   The RTC of the statement is $[2_R\ 1_A\ 0_M]$

4. **Pctr := Stack [Display [Dis-top + Inst-reg. adr. offset]]**

   The Inst-reg. adr. offset field of Inst-reg is known to the firmware and appropriate mask is known to the firmware for extracting the field. The transfers are:

   IR ---> ACC ;(R) Mask(offset) ---> ALU-2 ;(R)

   AND ;(A) ALU-0 ---> ALU-2 ;(R) Dis-top ---> ACC ;(R)

   ADD ;(A) Display [ALU-0] ---> ALU-2 ;2(R)

   Stack-base ---> ACC ;(R) ADD ;(A) ALU-0 ---> MAR ;(R)

   READ ;(M) MDR ---> PC ;(R)

   (Note: Offset field was assumed to be a right-aligned field, in cases of other fields, e.g. Inst-reg. adr.

Lex, the SHIFT operations required are counted as one average ALU transfer.)

The RTC of the statement is $[9_R\ 3_A\ 1_M]$

5. **Dis-top := Stack [Local-4];**

The variable local is available in a processor register 'Local' and the constant 4 is a constant coded in the micro instruction. The transfers are:

L ---> ACC ;(R) 4 ---> ALU-2 ;(R) SUB ;(A)
ALU-0 ---> ALU-2 ;(R) Stack-base ---> ACC ;(R) ADD ;(A)
ALU-0 ---> MAR ;(R) READ ;(M) MDR ---> Dis-top ;(R)

The RTC of the statement is $[6_R\ 2_A\ 1_M]$

The RTC's for the other statements are evaluated in a similar fashion.

The real transfer complexity (RTC) and the Average RTC (ARTC) of components discussed earlier are as follows:

1(a) RTC of $(M_1.CR = [87_R\ 27_A\ 11_M] + (n+1)[2_R\ 1_A\ 0_M]$

$$+\ n\ [10_R\ 3_A\ 1_M]$$

$$= [(79 + 12\delta_k)_R\ (24 + 4\delta_k)_A\ (10 + \delta_k)_M]$$

$$= [1\ \delta_k]\ .\begin{pmatrix} 79_R & 24_A & 10_M \\ 12_R & 4_A & 1_M \end{pmatrix}$$

& $\mathrm{ARTC}\ (M_1.CR) = [1\ \hat{\delta}_k\ ]\ .\begin{pmatrix} 79_R & 24_A & 10_M \\ 12_R & 4_A & 1_M \end{pmatrix}$

1(b) RTC $(M_1.BE) = [32_R\ 9_A\ 4_M]$ = ARTC $(M_1.BE)$

2(a) RTC $(M_2.CR) = [70_R\ 22_A\ 9_M]$ = ARTC $(M_2.CR)$

2(b) RTC $(M_2.BE) = [32_R\ 9_A\ 4_M]$ = ARTC $(M_2.BE)$

3(a) $$\text{RTC}(M_3.\text{CR}) = [1\ \beta\ ] \cdot \begin{pmatrix} 63_R & 22_A & 8_M \\ 13_R & 6_A & 2_M \end{pmatrix}$$

$$\text{ARTC}(M_3.\text{CR}) = [1\ \hat{\beta}\ ] \cdot \begin{pmatrix} 63_R & 22_A & 8_M \\ 13_R & 6_A & 2_M \end{pmatrix}$$

3(b) $$\text{RTC}(M_3.\text{BE}) = [1\ \nu\ ] \cdot \begin{pmatrix} 38_R & 13_A & 6_M \\ 15_R & 7_A & 3_M \end{pmatrix}$$

$$\text{ARTC}(M_3.\text{BE}) = [1\ \hat{\nu}\ ] \cdot \begin{pmatrix} 38_R & 13_A & 6_M \\ 15_R & 7_A & 3_M \end{pmatrix}$$

4(a) RTC (M_4.CR)

$= [57_R\ 17_A\ 7_M]$, if Proc. identifier is a global variable

or $= [60_R\ 17_A\ 7_M]$, if Proc. identifier is local to the calling block

or $$= [1\ \delta_k'\ ] \cdot \begin{pmatrix} 70_R & 20_A & 8_M \\ 10_R & 3_A & 1_M \end{pmatrix},$$

if procedure identifier is an intermediate variable.

$$\text{ARTC}(M_4.\text{CR}) = [1 \qquad\qquad] \cdot \begin{bmatrix} 49_R & 16_A & 6_M \\ 8_R & 1_A & 1_M \\ 11_R & 1_A & 1_M \\ (21 + 10\,\hat{\delta}_k)_R & (4 + 3\hat{\delta}_k)_A & (2 + \hat{\delta}_k)_M \end{bmatrix}$$

where η_x , η_t , η_y and $\hat{\delta}_k$ have their usual meanings as defined earlier.

4(b) $\text{RTC}(M_4.\text{BE}) = [30_R\ \ 9_A\ \ 5_M] = \text{ARTC}(M_4.\text{BE})$.

On comparing the ARTC's of the CR and BE components of the three mechanisms M_1, M_2, and M_3, it could be concluded , as in the previous section, that in evaluating the average transfer cost of a program it is sufficient to consider the mechanisms M_2 and M_4.

3.4 Overall Performance

As discussed earlier, to estimate the overall performance of an implementation technique for an average program, it is essential to estimate AVTC and ARTC of variable access as well. As far as the comparison of the mechanisms M_1, M_2, M_3 and M_4 is concerned, the AVTC and ARTC of variable access for M_2 and M_4 need evaluation.

The variable access (VARAC) component of a mechanism should be composed of two separate procedures for Read and Write access. As the ARTC/AVTC of both the sub-components are identical, only the Procedure for Read access (RVAR) would be described for both the mechanisms. The declarative part of the mechanisms M_2 and M_4 are not repeated and one more system Glovar declaration for the variable Read-Value is assumed. In the description of the real processor architecture, Read-value maps onto MDR.

The procedure RVAR from M_2.VARAC is as follows:

```
procedure RVAR /* for mech $M_2$ */
   Read-Value := Stack [Display[Inst-reg.adr.Lex] +
                    Inst-reg. adr. offset]] ; (6)
end Proc ;
```

The AVTC (M_2.VARAC) = 6 ;

The ARTC (M_2.VARAC) = $[11_R\ 5_A\ 1_M]$

The procedure RVAR for M_4.VARAC is as follows:

```
  Procedure RVAR /* for mech M4 */
  If inst-reg. adr. Dlex = 'Global'(3) =>
       Read-value := Stack [GP + offset]; (5)
     ||Inst-reg. adr. Dlex = 0 (3) =>
       Read-value := Stack [LP + offset] ; (5)
       DO Chain := Inst-reg. adr. Dlex ; (1)
             Ptr := Stack [LP-1] ; (4)
       WHILE Chain = 0 (3[p + 1])
       DO  Ptr := Stack [Ptr-1] ; (4)
             Chain := Chain - 1 ; (3)
       OD
       Read-value := Stack [Ptr
                     + Inst-reg. adr. Offset] ; (5)
       OD /* p = lexical level difference between
        the level of declaration and access
         of the intermediate var. */
       fi ;
endProc ;
```

The AVTC (M_4.VARAC) = $5 + \eta_1 .8 + \eta_2 .11 + \eta_3 [14 + 10\ \hat{\delta}_4\ ]$, where η_1, η_2 and η_3 are relative frequencies of global, local and intermediate variable access respectively. δ_4 = the mean lexical level difference between the level of declaration and level of access of an intermediate variable computed over a large number of sample programs.

$$\mathrm{ARTC}(M_4.\mathrm{VARAC}) = [1\ \eta_1\ \eta_2\ \eta_3\ ] \begin{bmatrix} 4_R & 2_A & 0_M \\ 10_R & 3_A & 1_M \\ 13_R & 3_A & 1_M \\ (22 + 10\ \ )_R & (5 + 3\ \ )_A & (2 + \ )_M \end{bmatrix}$$

For comparing the two mechanisms, as far as variable access is concerned, the preliminary statistics available in [Dep82b] are used. The statistics are:

$\eta_1 \cong 0.38$, $\eta_2 \cong 0.28$, $\eta_3 \cong 0.34$, and $\hat{\delta}_4 = 1.0$.

On substituting, $\mathrm{ARTC}(M_4.\mathrm{VARAC}) = [22.3_R\ \ 6.5_A\ \ 1.7_M]$

& $\mathrm{AVTC}(M_4.\mathrm{VARAC}) = 13.77$

DePrycker's Program activity characterization model would be used to evaluate the average cost of using an implementation technique [Dep82a]. The average VTC of a block structured program using the technique proposed by Rohl, could be represented as

$$\mathrm{AVTC}\ (P(M_2)) = n_b.24 + n_p.57 + n_v.6^{4},$$

---

[4] The term AVTC (P(M)) stands for Average virtual transfer complexity of an average program activity using mech M .

where n_b = mean number of Block Entry-Exit in a program

n_p= mean number of procedure Call-Return

n_v= mean number of variable accesses.

A comparison of the VTC of an average program execution using the two techniques indicate that

$$\mathrm{AVTC}\ (P(M_2)) < \mathrm{AVTC}\ (P(M_4))$$

provided -

$$24\ n_b + 57\ n_p + 6\ n_v < 24 n_b + (42.5 + 41.5\ \eta_v)\ n_p + 13.77\ n_v$$

or $\eta_v \leq 0.35$ (considering n_v/n_p to be a small positive quantity) .

[The preliminary statistics obtained in [Dep82b] from the nine numerical programs for digital filtering and speech recognition are used for these evaluations.]

• It should be noted that the observations regarding the effectiveness of using either of the techniques (M_2 and M_4) remain unaltered .

It is extremely difficult to predict the effectiveness of either of the techniques in terms of ARTC , unless some reliable statistics are available regarding n ,n ,n and η_v .

$$\mathrm{ARTC}(P(M_4)) - \mathrm{ARTC}(P(M_2)) =$$

$$n_p \cdot [31.5\ \eta_v - 11.5)_R (9\eta_v - 5)_A (5\ \eta_v - 2)_M] + n_b[-2_R\ 0_A\ 1_M] + n_v[10.2_R - 0.8_A\ 0.7_M].$$

The average execution cost of a component could be obtained by multiplying the ARTC matrix by a cost matrix (1 W_1 W_2) , where W_1 and W_2 are the cost of an average transfer through the ALU and the cost of a register-memory transfer respectively. W_1 and W_2 are the values normalized with respect to the cost of a transfer between two registers. The evaluation of the cost includes the technology dependence.

It has been shown in this chapter that it is possible to establish a methodology for choosing the hardware/firmware support for the variable addressing mechanism in the block structured HLL environment. The methodology could be extended for other similar architectural components.

The motivation underlying the study was to choose one of the four well known addressing techniques best-suited for design of the proposed architecture directed towards the HLL Ada. In absence of usage statistics for an Ada environment, the available statistics for Algol-60 were used. It was surprising to discover that the not so well-known technique suggested by Rohl yields such an effective mechanism. It might be noted that Rohl's technique is quite suitable as long as formal procedure parameters are not taken into consideration. It could be considered almost as good as Tanenbaum's technique for the Ada environment as Ada does not allow formal procedure parameters. In absence of any reliable statistics for η_v for Ada, it was not possible to come to a definite conclusion regarding the choice between Tanenbaum's and Rohl's techniques. But considering some of the preliminary statistics in [Dep82b], eg. $\hat{\beta}= 1$ and $\hat{\delta}_k = 2$,

$\eta_y \geq 0.5$ is quite high. In such a case, as far as AVTC of program activity in Ada is concerned, Rohl's technique appears to be superior.

Chapter 4

An Overview of the Proposed Architecture

The proposed architecture could be broadly characterized as a stack processor with tagged memory and capability based addressing and protection mechanisms. The rationale behind the use of capability based addressing and protection mechanism have already been established. In Chapter 2 the advantages of using a tagged memory approach and especially the advantage of using tagged capabilities for efficient domain switching have been explained. This chapter includes the rationale for choosing a stack oriented instruction set for the processor.

In presenting an overview of the architecture, the memory organization will be briefly indicated without any details of policies and mechanisms for virtual memory management. The primary emphasis in this chapter will be on the processor organization. The capability mechanism will be explained in detail with respect to the adopted principle of domain switching and implementation of abstract data types. But the discussion will not include details of the capability mapping mechanism or management of longterm capabilities in the secondary storage.

The chapter is organized into six major sections. The memory organization is introduced in the first section. The second section deals with the rationale behind the choice of the processor organization. The problems of protection in a stack processor organization are discussed along with the proposed solutions. The organization of the process stack segment is discussed in detail and the necessary hardware/firmware support for management of the process stack are specified in this section. The details of the capability mechanism are presented in the third section. The changes in the structure of the process stack segment due to the inclusion of protection domains and the capability mechanism are explained in this section along with the corresponding additional architectural supports. The fourth section includes the architectural support provided for dynamic type checking, parameter passing, accessing and representation of dynamic objects. This includes explanation of the representations of various primitive data types in the architecture. The principal object in the architecture, the packet object, is introduced in the fifth section. The details of the representation of this object and the interactions of the various fields of the packet with the activation records in the process stack are explained. The final section includes the description and explanation of the architectural support for implementing abstract data types. It could be observed from the above discussion that various architectural features are introduced in separate sections and the processor organization is systematically developed through the chapter.

4.1 Memory Organization

The virtual memory space is essentially visualized as three distinct sections:

a. Stack space,

b. heap space and

c. Packet space.

Another permanently resident segment for some resident operating system code for supporting memory mapped I/O and directory space for capability mapping information could be reserved in the Primary memory. (Details of memory management are not included in this thesis).

The stack space consists of segments representing the process stack segment of a process. Primary constituents of a process stack segment are the domain frames that include activation records for each domain. Further discussions on the domain frames and activation records of a process stack segment are included in sections 4.2 and 4.3. The process stack segment is protected by the capability register CR[10].

The heap space is meant for objects that are defined and created at run time of a program. Typical examples of such objects are dynamic arrays, discriminant records and accessed variables. Only objects that could be well defined at compile time but still occupy heap space are the composite objects (records, arrays, etc) that are returned as results of function subprograms in Ada. When an object is created by a process in the heap space, a reference pointer (a capability) is returned and stacked on the stack segment at the appropriate place. Thus every object in the heap space have unique system-defined names and are protected through the capability mechanism.

The packet space in the memory is allocated to packet objects. The description of the contents of the packet object is postponed until section 4.5. The packet object encapsulates a separately compiled module of Ada program, i.e., one or more subprograms, package or a task. Every packet object is protected by means of a capability.

The primary memory is word addressable. A preliminary specification of the word length is 64 bits which includes a 4 bit tag. The two hardware registers associated with the primary memory are the memory address register MAR and the memory data register MDR.

4.2 The Stack Processor

As noted earlier the proposed architecture could be broadly categorized as a stack machine. It is necessary to justify the choice of a stack oriented instruction set for an architecture directed towards Ada. A study of compiler systems for all the known programming languages, for the conventional register oriented architectures, reveals that no compiler treats the available hardware as a monolithic resource. Instead, some basic run time model for the architecture is adopted. The implementation techniques presented in the literature for any Algol-like language invariably assume a stack oriented machine as the abstract target architecture [RaR64, Gri71, Gri74, BaC79, BjO80, ShS80, Bar81]. The need and desirability for such an assumption has already been well established. Hence if the architecture under consideration represents a stack oriented instruction set, the need for simulating a run time model and thus generation of extra code for simulation cease to exist. There does not seem to be any disagreement among the computer architects regarding the usefulness of architectural support for efficient implementation and handling of activation stacks [Mye82, Dor79, Sto80, Ili82].

Although the basic run time model for any Algol-like language incorporates support for implementing activation stacks (for implementing scope rules, procedure calling conventions, etc.), that does not necessarily imply that the expression evaluations have to be done on the stack, through the generation of reverse polish code. Although efficiency consideration for expression evaluation is apparently a mundane issue, yet it is an extremely important issue. All assignment statements, IF and CASE statements, most DO statements and certain other statements in any procedural language involve expression evaluation. Studies indicate that 50%+ of written high level language statements and 75%+ of executed statements involve expression evaluation [Kee78, Tan78, Els76, Mye82]. Thus efficiency of expression evaluation significantly affects the code space requirement and execution efficiency of an architecture.

Two major studies on this issue of code space requirement and performance analysis of various instruction forms (with respect to expression evaluation) are due to Harvey Cragon [Cra79] and Glen Myers [Mye82]. Cragon's study is quite realistic and he observes that when a consistent set of assumptions are applied to the five instruction forms (2/3 address memory-memory, One Address Accumulator, register-register and 0/1 address stack-in-processor), no significant differences are found in either code space or performance. The conclusion is not too surprising from the information theoretic point of view.

Myers approaches the problem in a slightly different way in the sense that he measures the S and M measures for the above mentioned instruction forms by considering seven different expressions and their corresponding frequencies of occurrence in high-level language programs. The analysis indicated that 0/1 address stack-in-processor form is marginally inferior to 2/3 address memory-memory form with respect to the M-measure. In the context of the S measure, his analysis shows that the 2/3 address memory-memory form is definitely superior to the 0/1 address stack-in-processor form. The technique of the analysis is positively biased and unrealistic in the sense that unless realistic instruction streams are considered, 0/1 address stack form would require extra load-store sequences that are not really necessary. A similar argument holds for register oriented forms.

Based on the results of these studies, the conclusion is that the choice between the instruction forms (stack-in-processor, 2/3 address memory-memory) should not significantly affect the code space requirement or execution performance of the processor. The other advantages of a stack processor for an Algol-like environment have been indicated earlier, and thus it was decided that a stack instruction set would be the most suitable for an architecture directed towards Ada.

Myers [Mye82] points out in his report that though the stack-in-processor form seems to be reasonably attractive for expression evaluation, yet it is unrealistic to have the complete stack in the processor. A study by Tanenbaum [Tan 78] indicated that 99.7% of expressions in a typical program do not require more than four operands on the stack. Thus it is not unrealistic to assume that four top of the stack locations exist in the processor. Management of such a stack top does not really complicate the situation [Bla77, Den80, Sto80]. Moreover the use of a stack cache [Dit82] and instruction look ahead principles [Dit80] for stack machines have been adequately established in the literature. In the present proposal, four top of stack locations are considered to be available in the processor.

4.2.1 The Salient Features and Problems in Stack Architecture

Now that the suitability of a stack processor is established, the important features of a stack architecture will be summarized. Some of the problems associated to stack architectures will also be indicated along with the proposed solutions.

Some of the important features of the stack architectures are the following:

a. It allows easy implementation of block structured languages like Ada.

b. The usual register allocation problem for temporary variables is absent as the activation record of a procedure or a block provides automatic allocation of temporaries and local variables on the activation stack.

c. As indicated in (a), the stack architecture provides a natural support for allocation of procedure activation records and thus parameter passing becomes extremely simple. Similarly the protection domains of a process could be allocated on the process stack segment as domain frames. Thus capability passing across domains during domain switching becomes as straightforward an operation as that of parameter passing. This point will be further elaborated in the next section.

d. It provides locality of reference, as most of the operands and parameters are available in the stack segment. The stack segment of a process is protected by a capability that is made available in a predefined register at the beginning of a process and thus virtual address translation of objects existing in the stack segment does not require any additional overhead.

e. It allows stack relative addressing modes in which the necessity of explicit specification of the segment address (i.e., the capability for the segment) does not arise, thus yielding shorter instructions. On the other hand it provides compaction of code space through zero address instructions (where the top two elements of the stack are the implicit operands).

The concept of 'own' variable (OV) [Pra75] poses allocation problem in an stack architectures. Own variables declared in a procedure cannot be allocated on the activation stack as the variable should exist between activations of the procedure. In Ada, as mentioned earlier, variables declared in a package are to be treated as 'own' variables. The problem is solved in the proposed architecture by allocating 'own' variables, that are scalar or statically determined composite structures, in the OV space in the packet object. The own variables that are dynamically determined composite structures would be stored in the heap space with references to the heap objects in the OV space of the packet object, representing the Ada package. The solution is explained in detail in section 4.5.

Similarly, stack architectures do not provide any simple technique for implementing abstract data type mechanisms. The problem is not due to any deficiency in the stack mechanism but due to conflicting requirements of 'information hiding' and scope/visibility rules in block-structured languages. An elegant solution to the problem is presented in the proposed architecture through a combined interaction of the capability mechanism, the stack mechanism and the packet object. The details of abstract data type implementation is given in section 4.6.

A couple of protection problems are also associated with the commonly known stack architectures. One of the problems is linked with the display-relative addressing for block-structured languages. The concept of display relative addressing was discussed in Chapter 3. The protection problem associated with this addressing mechanism is that there is no check on the offset value that could be added to the contents of the display register for accessing an object in a particular lexical level. As there is no check on the limit on the offset value, any address on other lexical levels could be generated, although this is illegal from the point of visibility in block-structured languages. This problem is handled in the Burroughs series of machines through certified compilers [Dor79]. An easy and natural architectural solution to the problem is to provide a limit field in each display location.

Some machines allow generating addresses relative to the 'top-of-stack'(TOS) and the address is checked by the base bound pair for the whole stack segment. Obviously such explicit TOS relative addressing should not be permitted in any machine instruction (it could be done at the microcode level). Similarly most of the commercially known stack architectures do not restrict the POP instruction and zero address instructions to only the temporaries. Usually the issue of protection of the administrative information, parameters and local variables are relegated to the certified compiler. The proposed architecture takes a different stand and demarcates the base of the temporary area through a TB register; tags are provided for words of AIR area and the above mentioned instructions are only allowed to operate on temporary operands.

An important problem in all stack machines is the problem of ensuring the integrity of the procedure calling sequence. Again this issue is not usually considered as a problem as machine language programming is not allowed on most of these machines [Dor79, Org73] and the compiler generated code maintains the integrity. As indicated in Chapter 3, the proposed architecture takes care of the problem by microcoding the call sequence for procedure calling without parameters. For procedure calling with parameters, a different instruction pair is used (PBEGIN and PCALL). The details of the architectural support for calling Ada subprograms with parameters are provided in section 4.4. The scheme requires PBEGIN to precede a PCALL. The requirement is satisfied by a status bit at the microcode level. The microcode sequence for PCALL requires the status bit to be set (the status bit could only be set by a PBEGIN) for a successful execution.

4.2.2 A Preliminary View of the Stack Segment

Figure 4.1 presents a preliminary view of the stack frame and shows the various hardware registers related to the management of information in the stack segment. The view is preliminary in the sense that various other features and architectural supports will be added on to this view as we proceed through the chapter.

In Chapter 3, it was demonstrated that how the implementation technique chosen for designing the architectural support for variable addressing affects the execution efficiency of the architecture. In that chapter it was shown that the implementation technique based on Rohl's proposal was the most suitable one for designing the architectural support for variable addressing for an architecture directed towards Ada. Hence the view of the stack frame developed in this section and the corresponding hardware supports shown in Figure 4.1 are directly derived from the conclusion in Chapter 3.

In figure 4.1, the view of stack segment shows three activation records. The most recent activation of a block or procedure is represented by the top most stack frame.

The pointer to the base of the top most stack frame is contained in the register FM. Similarly the register TB points to the base of the area in the stack frame allocated for storing of temporary variables and expression evaluation. The lexical level index of the last activated block or procedure body is contained in the DIS-TOP register. The corresponding display register [Display[Dis-top]], in the bank of display registers, points to the base of the area in the latest activation frame that is allocated to the local variables of that activation. The concept of display register oriented addressing has already been explained in Chapter 3. The area allocated to the administrative information for return/exit (AIR area) contains information that is dependent on whether the frame under consideration represents a block or procedure activation record. If the frame represents a Procedure activation record then the AIR area contains the link, return address, previous Dis-top, previous FM and the previous TB. In case of a block activation record all the above information except the return address and the previous Dis-Top are present. The precise meanings of these AIR contents have been explained in Chapter 3 in the context of Rohl's mechanism (M_2). In the next section, once the capability mechanism in the architecture is introduced, this preliminary view of the stack segment will undergo substantial change to include frames for protection domains. Similarly after introduction of the parameter passing

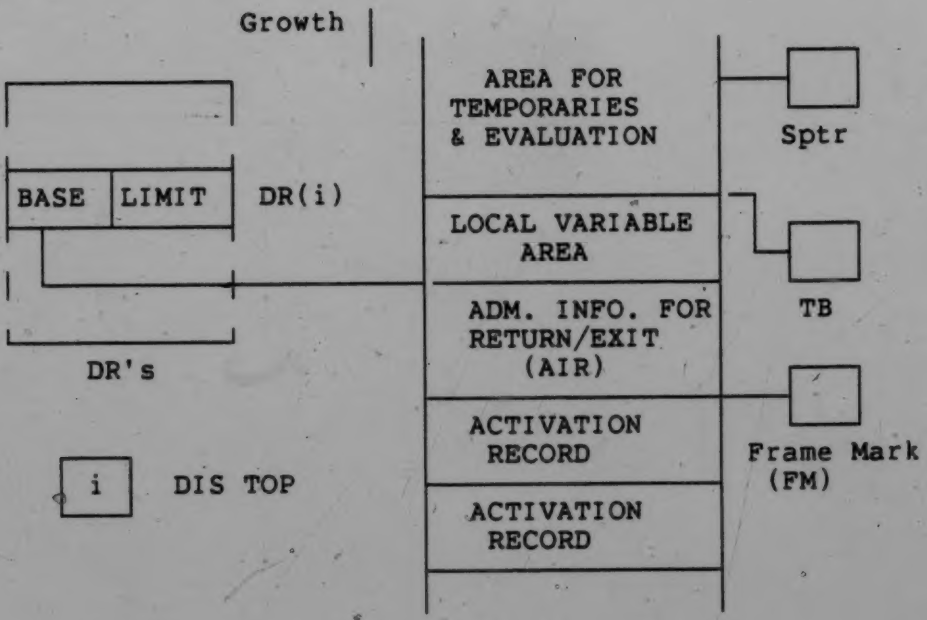

Growth
AREA FOR TEMPORARIES & EVALUATION
Sptr
BASE
LIMIT
DR(i)
LOCAL VARIABLE AREA
TB
ADM. INFO. FOR RETURN/EXIT (AIR)
DR's
ACTIVATION RECORD
Frame Mark (FM)
i
DIS TOP
ACTIVATION RECORD

Fig. 4.1 : A View of the Stack Frame

mechanism, the structure of the procedure activation record will be modified to reflect the necessary changes. Additional hardware/firmware supports will be introduced as and when necessary in the process of stepwise development of the complete processor.

4.3 The capability Mechanism

Before getting into the finer details of of the capability mechanism, in the proposed architecture, it is necessary to present an overall picture of the capability based addressing and protection as adopted for this design.

It was indicated earlier that any separately compilable unit in Ada (a subprogram, package or task) is represented in the machine level as a packet object in the packet space of the memory. A packet object consists of pure (reentrant) code corresponding to procedures, function or package initialization routines and some reserved areas for descriptor templates, own variables as well as capabilities. So an Ada process executes code segments of various packet objects associated with the process. In this architecture a packet object relates to a specific protection domain. Hence an Ada process could execute in a number of protection domains. There also exists a one-one correspondence between a packet object and section of the process stack segment named as a domain frame.

The activation records related to a packet object (corresponding to procedures and blocks) are contained in the domain frame associated with the particular packet under consideration. The stack segment of a process and every packet object is protected by the capability mechanism. A capability (in other words, a unique system-defined name) would be associated with a packet object when it is created by a process. The process stack segment gets associated with a capability when the corresponding process is initiated. Further details on the packet object and the interactions between a packet object and the process stack segment are postponed until section 4.5.

In this proposal, capabilities name and control access to objects in the virtual address space. The capability associated with an object represents a unique system-wide name for the object irrespective of where in the system the object is currently located and regardless of which process uses it. The actual representation of an object is not defined in the architecture (as in SWARD [Mye82]). Some typical objects that are associated to and identified by a capability are - process stack segments, packet objects, simple and composite structures in the heap space. Each object in the system is assigned the unique name at the time of creation (at run time).

As the addressing mechanism of the architecture is built around the concept of capability based addressing, discussed earlier, the virtual address of an operand is considered to be of the form (capability, offset). It would be seen in the discussions to follow that often it will not be necessary to explicitly specify the capability component of the virtual address. The form of virtual operand address depends on the type of instruction referring the operand. For the stack oriented instructions, referring to operands in the process stack segment, the virtual address translation mechanism would automatically assume the capability of the process stack as the implicitly declared capability in the operand address. Similarly instructions specifying address references in the instruction space of a packet object would be implicitly specifying the capability associated with that packet. Capabilities have to be explicitly specified only for the operands existing in the heap space.

4.3.1 The capability Representation

Before describing the exact form of the capability proposed in this architecture, it is necessary to enunciate the rationale underlying the design decisions. Consider the requirements of the capability representation:

1. It should represent an unique system wide name for an object (process stack segments, packets and objects in the heap space).
2. Access rights for operations permitted on the object should be specified in the capability representation.
3. The capability representation should be able to uniquely identify a protected procedure in the packet object. (It is particularly necessary for implementation of

abstract data types in Ada.)

4. Similarly it should be able to name and restrict access to a word within a composite object without exposing the complete object. This requirement arises while passing element of an array or a record as a reference parameter or while passing a word in the stack segment across a domain frame. A further restriction regarding a capability identifying a word in the process stack segment is that the capability should not be allowed to be copied out of the stack segment.
5. The name/identifier part of the representation should be protected against any modification. It should be possible to modify the access rights to the extent of deleting certain rights but no amplification should be permitted.

The first requirement implies that enough bits should be allocated in the capability representation to uniquely name objects created throughout the lifetime of the system. There are various suggestions in the literature regarding the number of bits required for the unique object identifier. Some of the suggestions are:

Linden - 50 bits [Lin76], Myers - 30 bits [Mye82], Dennis - 37 bits [Den80].

Dennis has shown that even with a highly inflated assumption of 36 objects being created per second, a 37 bit identifier should be able to provide unique names for 125 years! In the present design 36 bits are reserved for the capability identifier.

The access authorization field would have five bits representing the five different authorizations -'Read', 'Write', 'Enter', 'Copy' and 'Destroy'. A process possessing a capability for a segment with 'Read' authorization would only have the authorization to read the segment pointed by the capability identifier. Similar is the 'Write' authorization. The 'Enter' authorization does not carry its usual meaning as in the C-list oriented capability architectures [Den66, Eng72, Nee74]. The 'Enter' authorization could only be present in a capability thatnames a packet object. The presence of 'Enter' authorization in a capability allows the process possessing the capability to switch its protection domain to execute in the packet specified by the capability. The 'Copy' authorization in the capability allows passing a copy of the capability across domain frames. The 'Destroy' authorization implies the authority to destroy the segment named by the capability. The capability representation for this architecture is shown in figure 4.2.

tag	form	authority	identifier	offset or index

Figure 4.2 The capability representation .

The requirements 3 and 4 introduce the necessity of having a 'form' field in the representation. This field as well as the 'Authority' field are not encoded for execution efficiency. The interpretation of the form field is as follows:

form : 001 : capability for an object segment and the offset field is to be ignored.

form : 010 : capability for a word in any segment other than process stack segment and the offset field represents the offset of the word in the segment.

form : 100 : Similar to 010 but the named word is in the process stack segment (cf. requirement 4)

Another form is 'all zero' to represent the capability for a packet, the presence of 'Enter' authority is mandatory to execute instructions in the packet. For this form with 'Enter' authority, the last 16 bits represent the entry point index in the header section of the packet object (in this case the field is not encoded and should only have single 1 at a time). Thus it is possible to have 16 protected entry points in the packet object. Futher details of the representation of the packet object are given in section 5.5.

The capabilities of the forms '010', '100' and '000' could only be created by a process having a capability of the form '001'. The creation of these three capability forms from the '001' form is handled by the COMPUTE CAPABILITY instruction.

4.3.2 Capability Mapping Mechanism

The proposed capability mapping mechanism and register support structure (in the next subsection) for capability based addressing bear resemblances to the corresponding architectural features in the Plessey 250 system [Eng72] and a capability architecture proposed by Dennis [Den80]. Thus details of the mapping mechanism or long term management of capabilities in the secondary storage will not be presented. Only so much detail will be provided as is necessary to explain the remaining new features of the architecture.

To implement capability addressing, it is necessary to map the virtual addresses (the capability identifier, offset pair) into primary memory addresses. The mapping mechanism could be very similar to those proposed for mapping segment descriptors in segmented memory management systems

[MaD74,Wil72]. A table is needed in the secondary storage to relate the capability identifiers to the objects stored in the backup secondary storage. As the table would be extremely large it could be represented as a tree of tables. A table is also required in the nonrelocatable part of the primary memory to contain information relating capability identifiers to primary memory addresses. This table will be referred to as the PMT (Primary memory table). The primary memory addresses will be present in the (base, limit) form.

An entry in the PMT would contain the following fields:

a. the capability identifier,

b. the 'base' of the object in primary memory,

c. the 'limit' or the length of the object,

d. some bits necessary for memory management and replacement polices and,

e. a pointer to the corresponding entry in the mapping table in secondary storage(to be used when segment fault occurs for the object linked to this entry).

Thus the PMT is used to map capability identifiers into primary memory addresses and the PMT would keep a record of the segments (objects) currently being used by a particular process executing in the system.

4.3.3 The Set of capability Registers

The architecture includes ten capability registers CR[1] - CR[10] to support the capability mechanism. These registers would contain the mapped form of the capabilities used in a specific domain. The set of registers is always associated with a particular domain of protection. The use of registers for mapped capability information is motivated by the desire for short addresses and fast referencing of capabilities that are often used. It was indicated earlier that when a capability is used for addressing, the mapping mechanism is invoked for retrieving the primary memory address (base-limit pair) from the PMT entry corresponding to the capability identifier under consideration. It is obvious that it would not be wise to use the mapping mechanism for generating each and every address. The capabilities that are to be used frequently, are loaded into capability registers (in the mapped form). In this architecture every address specifies a capability either implicitly (e.g., capability for process stack or packet object) or explicitly. The descriptions of the operations involving the process stack segment with domain frames and association of the set of capability registers with a domain frame are presented in the next section.

The format of the contents of a capability register is shown in Figure 4.3.

L	•••	authority	base	limit

Figure 4.3 Capability register format.

The instruction LOAD CAP REG loads a specified capability into the register indicated in the instruction. The LOAD CAP REG instruction invokes the capability mapping mechanism to retrieve the capability from the PMT and loads the base-limit fields of the PMT entry into the specified register. It also loads the 'Authority' field of the register from the capability and sets the load indicator (L) on. There could be some more bits in the capability register for management and replacement policies.

Of the ten capability registers provided in this design, CR[1] to CR[8] could be loaded and used by compiler generated code for user programs. The registers CR[9] and CR[10] are reserved for specific system use. The register CR[9] gets loaded with the capability for the segment, representing a packet, from which the instructions in a particular domain would be fetched and executed. The content of register CR[9] would get changed when domain switching takes place via an ENTER instruction. The program counter is always interpreted relative to CR[9]. The capability register CR[10] is always loaded with the capability representing the process stack segment. It is changed only when the processor switches from one process to the other. This thesis essentially deals with a single process and thus any changes in the content of CR[10] would never be encountered.

It should be observed that only the machine instruction LOAD CAP REG could write into the registers CR[1] to CR[8], that also by an explicit specification of a capability stored either in the packet or the process stack segment. There is no way in which any user-generated instruction could modify a capability, other than deleting some of the access rights (authority).

In this architecture, the execution of a process in various protection domains is synonymous to the execution of a process in various packets. This concept introduces an interesting notion of user-defined domains of protection. The term 'user-defined' is used as the user could define the contents of a packet object. The number of subprograms constituting the instructions in a packet object could be controlled by the user through separate compilation. This feature could be naturally visualized in the context of Ada, as Ada presents well defined means of separate compilation [GoH80].

When any procedure or block in a packet is executed, an activation record is created on the stack for the particular invocation. To define an encapsulation of the activation records of a specific packet object, the concept of domain frames in the process stack segment is introduced. The introduction of domain frames in the process stack segment changes the preliminary view of the stack frame presented in section 4.2.

It was mentioned earlier that the contents of the set of capability registers CR[1]-CR[9], at any instant of time, are uniquely related to a specific protection domain i.e., a domain frame on the process stack. To implement this relationship, a specified section in the beginning of the domain frame is reserved for the nine capabilities that could get loaded into the set of capability registers. When a capability register is loaded, the corresponding capability is stored in the reserved location corresponding to the particular capability register under consideration. This reserved area in the domain frame would be referred to as the capability register backup area (or CRB). Locations in the CRB area CRB[i] would represent the capability whose mapped version exists in CR[i] (for i =1...9). It should be noted that information in CR[i] is valid only when the load indicator in the register is ON. If CR[i] contains valid information, the semantics of LOAD CAP REG instruction implies that CRB[i] contains the original capability that has been processed by the mapping mechanism and the mapped version (cf. figure 4.3) exists in CR[i].

At the time of domain switching, the contents of the capability registers need not be stored in the new domain frame. It is sufficient to store the information regarding the status of the L indicators for each of the nine registers. While returning back to the previous domain, the status of L indicators are checked and corresponding capability registers are loaded from the information stored in the CRB area.

4.3.4 Introduction of Domain Frames in the Process Stack

A modified view of the process stack segment is shown in figure 4.4. The modifications to the earlier version presented in figure 4.1 are due to the introduction of the notion of protection domains in the architecture. The necessary hardware added on to the previous view for supporting this feature are indicated in the figure 4.4 and will be explained in this section.

The stack segment shown in figure 4.4 consists of three domain frames. It indicates that the process under consideration has already executed in two packets and has entered the third packet pointed to by CR[9]. A domain frame would consist of as many activation records as the number of procedure or block activations during the execution of the process in the corresponding packet. The third domain frame contains three activation records. The CRB register points to the base of the CRB area of the top-most domain frame.

The major modifications in information content of the AIR region of the first activation record in a domain frame is quite significant. The additional information in the AIR of the first activation record in a domain frame as compared to other AIR's (given in section 4.1.2) are the following:

a. contents of CRB register of the last domain,

b. complete display register bank and

c. L indicators of the nine capability registers from the previous domain.

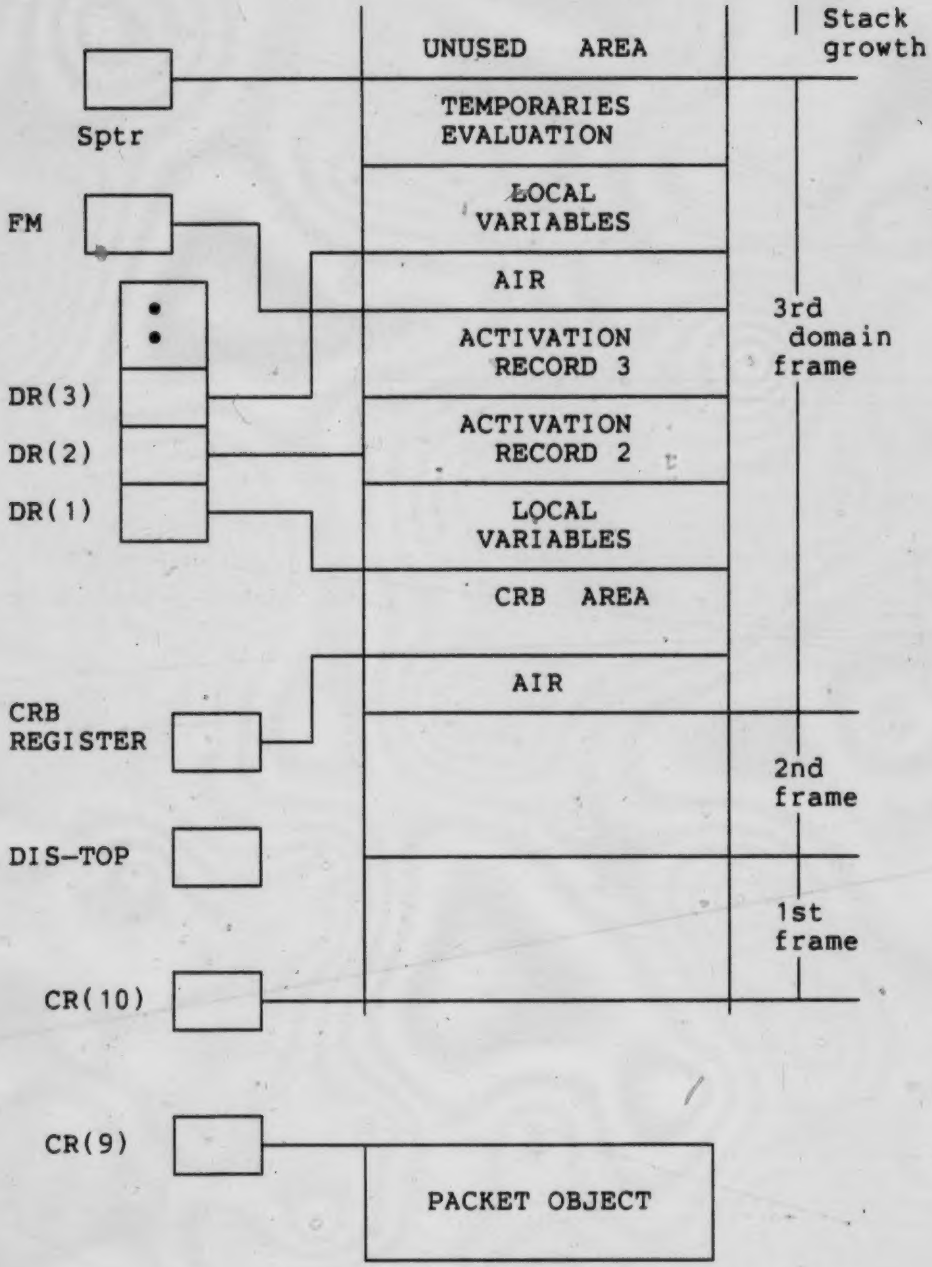

Stack growth
UNUSED AREA
Sptr
TEMPORARIES EVALUATION
FM
LOCAL VARIABLES
AIR
3rd domain frame
ACTIVATION RECORD 3
DR(3)
ACTIVATION RECORD 2
DR(2)
DR(1)
LOCAL VARIABLES
CRB AREA
AIR
CRB REGISTER
2nd frame
DIS-TOP
1st frame
CR(10)
CR(9)
PACKET OBJECT

Figure 4.4 : A Modified View of the Process Stack

The emptying of the display register bank during the domain switch prevents any access of activation records in the calling domain from the newly entered domain. Thus lexical level addressing implemented by display registers deals with the top domain frame only. Similarly tags on every word in the AIR [cf. section 4.5] allow only the RETURN group of instructions (RETURN,PRETURN,FRETURN) to use the contents of the AIR area, and prevent any other instruction from accessing information in the previous domains.

A new domain is entered by executing an ENTER instruction. The ENTER instruction requires a capability (with 'enter' authority) for the new packet object to be entered. The ENTER instruction expects the capability to be on top of the stack. So the calling domain prepares a capability by COMPUTE CAPABILITY instruction before issuing an ENTER instruction. The COMPUTE CAPABILITY instruction requires a capability (with 'enter' authority) in the packet object containing the instruction and a 16 bit value representing the entry point on top of the stack. On successful execution it leaves a capability with entry point index specified in the index field of the capability. In addition to storing the required information in the AIR and resetting the capability registers CR[1] to CR[9] of the first activation record of the new domain frame, the ENTER instruction does the following:

1. reserves space for CRB area,

2. loads CRB[9] with the capability specified in the ENTER instruction,
3. invokes the mapping mechanism and loads CR[9],
4. resets the display register bank and sets Dis-top to 1, and
5. loads the program counter with the address of procedure obtained from entry point information (corresponding to the index in the capability) available in the header of the new packet.

The return from a domain back to the calling domain takes place through the execution of one of the usual return instructions (RETURN or PRETURN). The difference between a usual return from a procedure and a domain switch is detected at the microcode level from an indicator in the AIR area associated with the activation record of the procedure executing the return. A domain return sequence performs the following in addition to the usual return sequence:

1. resets the capability registers CR[1] to CR[9],
2. executes a sequence of load capability register operations according to the state of L indicators in the AIR of the returning domain frame, (note: only microcoded sequences are allowed to access information in two different domain frames) and
3. reloads the display register bank and the Dis-top register with values stored in the AIR of the returning domain frame.

4.4 Some Special Issues in an Architecture for Ada

Architectural support for efficient implementation of abstract data types and the details of the representation of the packet object will be dealt with in the next two sections. In the following subsections, other architectural features will be presented that are specifically designed for efficient runtime representation of Ada programs.

4.4.1 The Primitive Data Types

The usefulness of a tagged memory representation was indicated earlier. This architecture uses short tags of 4 bits, with every data word in the memory. The interpretations of the tag fields are as follows:

Tag	Type of information
0000	Undefined word
0001	A word representing real number
0010	An integer word with pointer to a rangeword
0011	An integer word with lower bound of 0
0100	An integer word with lower bound of 1
0101	An AIR word
0110	An instruction word
0111	A capability word
1000	A rangeword
1001	A word in packet header
1010	A word in array descriptor
1011	A boolean word

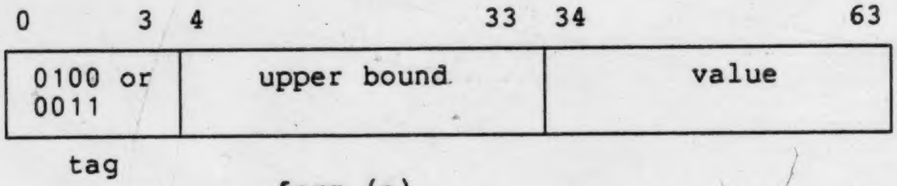
The rationale behind the above mentioned primitive types will be discussed in the following sections.

4.4.2 Support for Subranges and Constraint Checking

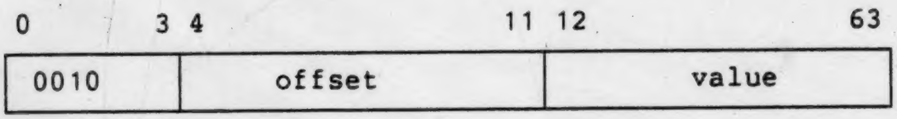
Studies done on occurrence of variables in programs put array access at less than 15% whereas simple variable access accounts for 60% of the total [Tan78, Rot76]. In the pre-Pascal languages, accessing a simple variable was a straightforward operation. However introduction of the concept of subranges demands additional architectural support for constraint checking at run time. Most of the well known commercial architectures do not have any special support for efficient constraint checking and usually generate significant amount of additional machine code for variable accesses to satisfy such requirements in the programming languages. This is definitely not acceptable if some convenient architectural features could easily improve the execution of programs in language like Ada or Pascal [Bis80, Hil81]. So it was decided to provide an efficient architectural support for constraint checking for integer variables and a reasonably good (better than say, VAX 11/780 or IBM 370) support for real representations. Checking of subrange is synonymous to checking array indices using descriptors.

The above observations lead to the definition of primitive data types with tags 0010, 0011 and 0100 for representing integers along with subrange information. Empirical studies have shown that zero and one tie for the first place as lower bounds and together they cover the majority of subrange definitions for integer types [Bis 80]. So two primitive data types are defined for integer representations that implicitly encode the lower bound information in the representation.

The format of the integer words with the tags 0010, 0011 and 0100 are shown in figure 4.5.



form (a)



form (b)

Figure 4.5 Integer Words

The tag 0100 implies that the lower bound is 1 and the upper bound is a positive value and is contained in the 'upper bound' field (bits 4-33). Similarly tag 0011 implies that the lower bound is zero and the positive upper bound is contained in the upper bound field. These two formats are used for short integers with positive upper bounds. The form(b) is used for longer integers and for any integers that can not be represented by form(a) (i.e., when lower bounds are not zero or one and also when lower upper bounds are not positive). The offset field in the form(b) represents an offset in the descriptor template area in a packet, where a rangeword with tag 1000 is stored. The format of a rangeword is shown in figure 4.6.

1000	upper bound	lower bound

Figure 4.6 A rangeword.

The upper and lower bound fields would contain values with sign information. A single rangeword could be used to represent the subrange information for many variables if they all have the same subrange constraint. It could be argued that the representation in form (b) introduces unnecessary waste of memory space. The use of the system-wide fixed word length concept always introduces the problem of lack of proper utilization of the available memory space, but the use of fixed word length improves execution efficiency. If the usefulness of the fixed word length concept is assumed, then form (b) representation actually improves the space utilization in the sense that the space used for the range information would not have been used by a short integer anyway.

The advantage of associating the subrange information directly with the variable is that it will not be necessary to fetch and execute separate machine instructions for loading range values and perform range check operations for every variable associated with subrange information.

Some extra hardware facilities are also proposed in this design to facilitate the often used operation of range checking. Two extra hardware registers UR and LR are provided in the ALU that would usually contain the upper and lower bounds of the variable representing the destination of the ALU result. The ALU would also include two separate comparators, for the purpose of comparing the ALU-0 to the values stored in UR and LR registers. A special flag RC in the ALU is used by the microcode for enabling and disabling automatic range checking.

There are essentially two instructions in the proposed architecture that deal with the special registers and the automatic range checking hardware. The LOAD RANGE (LR) instruction is used for real operands and the range is explicitly specified in the instruction. The LOAD RANGE instruction loads the explicity specified upper and lower bound values into UR and LR registers respectively and it also sets the RC flag. When the RC flag is set, the gating of the ALU output to the destination is delayed until the range check is automatically performed. The LOAD RANGE instruction is generated by the compiler only when constraint checking on the ALU-output is necessary.

The other instruction that invokes automatic range checking is the CSTORE (check before storing) instruction. The address of the destination location is specified in the CSTORE instruction. The microcode sequences to be used by the CSTORE instruction depend on the tag of the destination location. In cases of 0100/0011 tags, the UR register is loaded from the upper bound field of the variable and 0 or 1 is loaded into the LR register. In case of 0010 tag the register UR and LR are loaded from the location in the packet specified by the offset field of the destination location. Automatic range check is performed before the storing operation.

4.4.3 Support for Parameter Passing in Ada

Implementation of the parameter passing convention in Ada has to deal with some new problems that were not present in earlier block-structured languages like Algol, Pascal or PL/1. The semantics for scalar parameters imply that

1. Any range constraint on the formal parameter, for an IN or INOUT parameter, must be satisfied by the actual parameter at the beginning of the call.
2. Any range constraint on the variable which is the actual parameter, for an INOUT or OUT parameter, must be satisfied by the value of the formal parameter upon return from the subprogram.

The above two conditions require constraint checking (with respect to the corresponding formal parameters) after evaluation of the actual IN or INOUT parameters and constraint checking (with respect to the actual parameter) before the assignment of the formal OUT or INOUT parameters to the corresponding actual parameters. The problem lies in the fact that at the point of return the actual value of the local variable (representing the formal) corresponding to OUT and INOUT parameters must lie within the range constraints of the actual parameter. Such a check cannot be performed within the procedure body as the range constraints of actual parameters can not be known to the procedure.

None of the known architectures provide elegant solution for this problem. The proposed architecture provides an unconventional but efficient support to deal with this problem. Two new pairs of instructions are provided PBEGIN-PEND and PCALL-PRETURN.

The PBEGIN-PEND pair works in the same way as the ENTRY and EXIT instructions used for block entry and exit (discussed in Chapter 3) except in that PBEGIN sets a flag (PB) at the microcode level.

The PCALL instruction is used for calling procedures with parameters. A virtual block is built around the procedure body and all parameter type checking operations are done in this virtual block. The entry to the virtual block is caused by the execution of the PBEGIN instruction. The parameters for the ensuing PCALL instruction is prepared in this virtual block and left on the process stack. There are some interesting problems related to the management of the display registers at the time of entry to the procedure body. It could be best explained through an example. Assume the following conditions:

(a) the procedure is called from lexical level 4; and

(b) the procedure identifier is declared at lexical level 2 and thus the new value in display register 3 would be pointing to the section of the stack segment that would be allocated to the procedure body after execution of the call instruction.

At the point of calling, the variable accessing environment is represented by the Display registers 1 to 4. The type checking for IN and INOUT parameters have to be performed in this environment as the actual parameters are accessible only in this environment. On execution of the call instruction, the variable accessing environment for the procedure body is represented by display registers 1, 2 and 3(with the new value). Thus the actual parameters evaluated in the old environment would not be accessible in this new environment. A similar situation occurs after execution of the return instruction. The type checking of the formal OUT and INOUT parameters against the actual parameters cannot be performed in the environment represented by the display registers 1, 2 and 3, as the actual OUT and INOUT parameters may not be accessible in this new environment.

The introduction of a virtual block enclosing the procedure body and a different way of handling the Dis-top pointer by these new pairs of instructions solve the problem. On execution of PBEGIN a new block at level 5 is created. The constraints on the formal parameters in the procedure are known to the calling level at compile time. The code for constraint checking is included in the virtual block invoked by PBEGIN. It should be noted that the variable accessing environment as available to the virtual block is the same as that of the calling level. So any variable declared in this environment could serve as an actual IN, INOUT or OUT parameter. The constraint check before executing PCALL is done for IN and INOUT actual parameters. After the check the actual IN or INOUT parameters are available on the stack at lexical level 5 of the calling environment. On execution of a call instruction, the variable accessing environment would change (as indicated earlier). The proposed mechanism requires that lexical level 5 of the calling environment be merged with the lexical level 3 of the new environment. This is achieved through the new instruction PCALL. Similarly, at the time of return, PRETURN would do the appropriate adjustments so that the procedure returns to the virtual block at lexical level 5 of the calling environment, but would still be able to access the formal parameters. The constraint checking for the OUT and INOUT formal parameters and assignment to the actual parameters are done by the code in the virtual block. Finally PEND is executed and the original calling environment is reestablished. S*A descriptions of the PCALL and PRETURN components for Rohl's mechanism (M_2) are given below. The declarative part of the mechanism M_2 is assumed.

```
Proc M₂.PCALL
   Stack [Sptr] := FM ; FM := sptr;
   Sptr := Sptr + 1;
   Stack [sptr] := Dis-top ; Sptr := Sptr + 1;
   Stack [sptr] := Pctr ; Sptr := Sptr + 1;
   Dis-top := Inst-reg.Adr. lex
   Pctr := Stack [Display [Dis-top] + Inst-reg. offset]
   Sptr := Sptr + 1 ;  Dis-top =: Dis-top + 1 ;
   Stack [sptr] := Display [Dis-top]
   Sptr := Sptr + 1 ;
   Display [Dis-top] := Display [Stack[Sptr + 3]];
   /* Assuming the AIR area to be same as described for
      Rohl's mechanism in chapter 3 */
end Proc

Proc M2.PRETURN
   Sptr := FM ; FM := Stack [sptr] ;
   Display [Dis-top] := Stack [Sptr + 3] ;
   /* the LINK is loaded from AIR  */
   Dis-top := Stack [Sptr + 1] ;
   Pctr := Stack [Sptr + 2] ;
end Proc
```

The parameter passing mechanism introduced in this section requires further modifications of the view of the stack segment shown in figure 4.4. A view of the procedure activation record with parameters, after the execution of the PBEGIN-PCALL sequence, is shown in figure 4.7. After execution of an usual CALL instruction, the Frame-mark (FM) and Display [Dis-top] would have pointed to the positions shown by (**) and (*) respectively, whereas, on execution

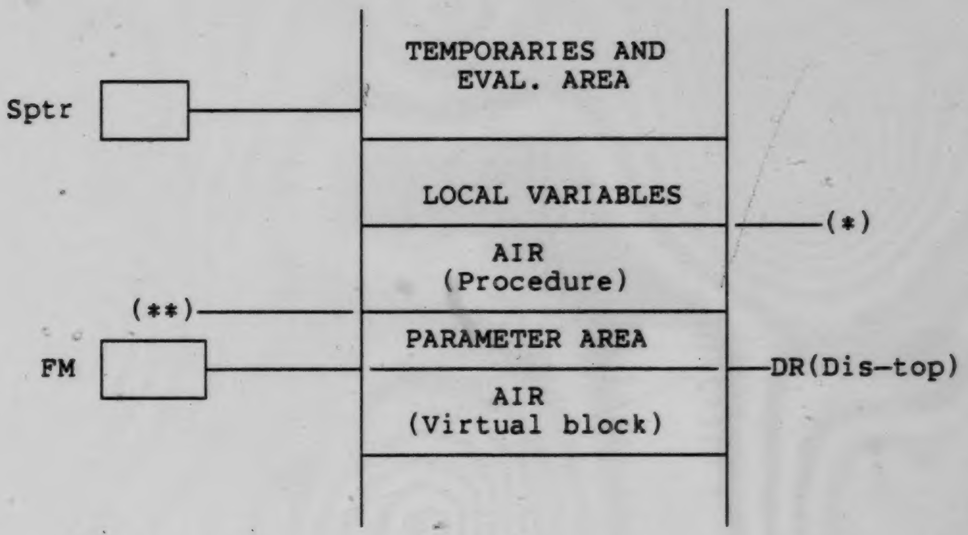

TEMPORARIES AND
EVAL. AREA
Sptr
LOCAL VARIABLES
(*)
AIR
(Procedure)
(**)
PARAMETER AREA
FM
DR(Dis-top)
AIR
(Virtual block)

Fig. 4.7: A Procedure Activation Record With Parameters

of a PCALL, a portion of the frame allocated to the virtual block gets included in the procedure activation record. It should be observed that the above mentioned modification do not cause any new protection problem as every word in the AIR area are tagged. The only modification necessary would be in the virtual address generated by the compiler. The offset parts of the (lexical level, offset) form of addresses for the local variables have to be modified to take care of the additional predefined offset introduced by the procedure AIR area.

4.4.4 Support for Dynamic arrays and Discriminant records

Arrays and vectors are represented in this architecture in the packed form [Tan76, BaC79] and are addressed using dope vectors. The records are flattened out and each field is treated as a separate variable [JeW72]. Handling of dynamic arrays and discriminant records deserve special considerations. The dynamic nature of discriminant records is due to the presence of a dynamic array as a field of the record. Hence discussion in this section will be restricted to representation and addressing of static and dynamic arrays. The rank of a dynamic array as well as the type of the elements are always known at compile time. The bound information for one or more dimensions could be specified at the execution time.

The statically defined arrays and array field of a record are allocated on the activation stack. The elements of a dynamic array or array component of a discriminant record are allocated in the heap space with a capability for the representation in the dope vector in the corresponding activation record in the process stack segment.

The packed representation (in column major form) of a static array A along with the form of the dope vector is shown in figure 4.8. The array shown in the figure corresponds to the array declaration:

A: array (INTEGER range $b_1..u_1$,...,INTEGER range $b_n..u_n$) of INTEGER;

where b_i and u_i $(i = 1..n)$ represent the lower and upper bounds of the dimension i respectively. The length l_i shown in the dope vector is -

$$l_i = u_i - b_i + 1$$

The address of an array element is obtained by the instructions INDEX and SINDEX. Both the instructions require the rank of the array, the subscripts of the element and the address of the pointer word in the dope vector to be available in a predefined sequence on the activation stack. The INDEX instruction does range checking for each and every subscript before generating the address, whereas the SINDEX (Safe Index) instruction is generated by the compiler when the subscripts are known to be correct. Proper use of such an instruction could lead to significant reduction

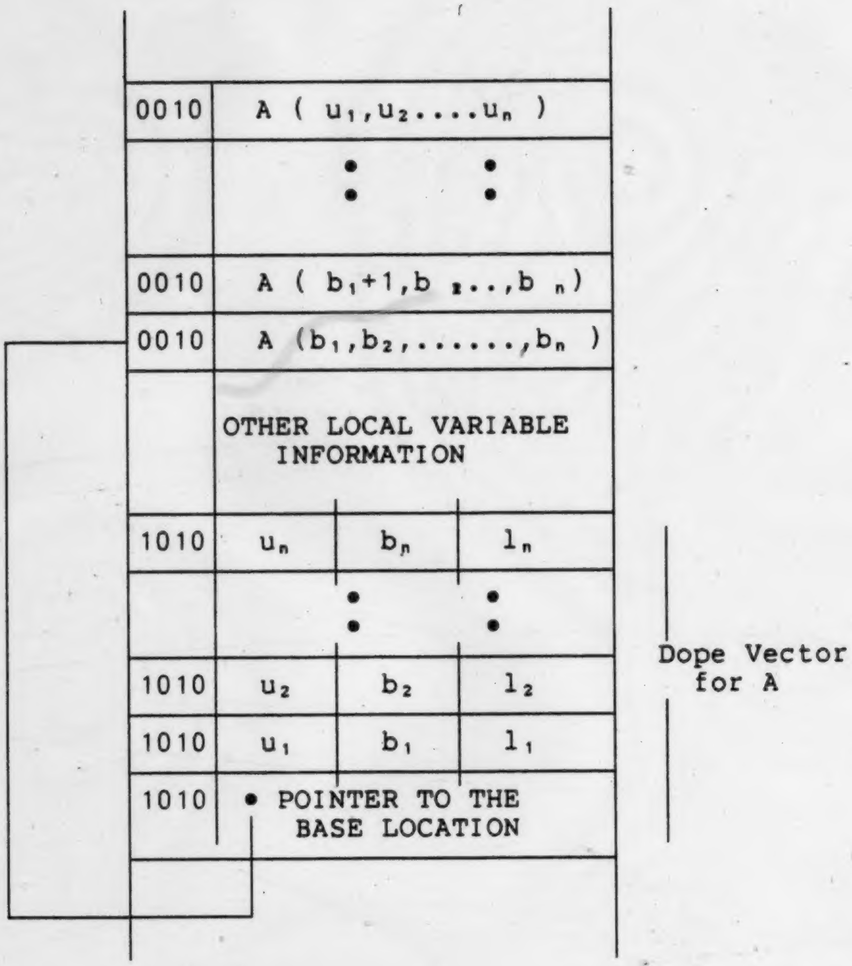

0010
A (u₁,u₂....uₙ)
0010
A (b₁+1,b ₂..,b ₙ)
0010
A (b₁,b₂,......,bₙ)
OTHER LOCAL VARIABLE INFORMATION
1010
uₙ
bₙ
lₙ
1010
u₂
b₂
l₂
1010
u₁
b₁
l₁
1010
POINTER TO THE BASE LOCATION
Dope Vector for A

Figure 4.8 : Array Representation

of execution time for array element addressing [BiB81].

Another interesting feature in the proposed design that facilitates representation of static and dynamic arrays is the dope vector template or descriptor template in a predefined space in the packet. It is possible to create a dope vector template in the packet object at compilation time as the rank of any array is always statically determined. The dope vector template is similar in structure to the corresponding dope vector that would be eventually created on the process stack. The difference between the template and the actual dope vector is that the template might have fields that cannot be specified at compile time and the pointer field is kept undefined. The dope vector template (or any other descriptor template) proves to be extremely useful for fast creation of dope vectors on the activation stack. The dope vector frame along with the statically known fields could be copied on to the stack by a BLOCK MOVE (BMOVE) instruction. Three instructions LDV1, LDV2 and LDV3 are provided for easy loading of the three fields of a dope vector word with the contents available on top of the stack.

Ada allows formal array parameters with one or more unconstrained dimensions. The dimensions derive their range constraints from the actual parameter. The mechanism of array parameter passing is also easily handled in the architecture by block copying (by BCOPY instruction) of the dope vector of the actual parameter onto the dope vector of the formal parameter. The actual array is re-created on the heap space and the capability for that area is stored in the pointer space in the dope vector for that array. It is important to note that the pointer space of the dope vector for a dynamic array always contains a capability to the actual array representation in the heap space. The same is true for dynamic fields of discriminant records.

4.5 The packet Object

The packet is the most important object in this design as it represents the output of the compiler for any compilable unit in Ada. The composition of the packet object proposed in this design appears to be quite similar to the module object in SWARD, at first glance. But a closer look reveals that they are quite different. The packet object is composed of the following areas:

a. The header information area (HIA) contains a list of indexes to the starting point of different areas in the packet object. It also contains a list of indexes to protected entry points in the instruction space of the packet object. Any word in this area carries the tag 1001.

b. An area OVS is allocated in the packet for storing own variables. The own variables of the scalar type exist in this space and the descriptor/dope vector templates for own variables of composite structure are stored in this space. Such variables are created in the heap space and initialised by the code generated for package initialisation.[5] When such an object is created in the heap, a capability for the object gets stored in the space allocated in the descriptor template (in the packet) for the object.

c. A space in the packet object (DTS) is allocated to descriptor/dope vector templates for arrays and records used by the instructions in the packet. The corresponding arrays and records are created in the stack segment when they are declared in procedures or blocks, using the templates stored in this space. This space would also contain the rangewords (described in Section 4.4.2) and the constants.

d. A space (CS) is reserved for storing capabilities to other packet objects. The capabilities stored in this space determine the authority to enter other packet objects. The 'use' clauses in Ada are translated as CREATE PACKET instructions. Execution of CREATE PACKET instruction in the initialisation routine of the packet object creates a packet object named in the instruction and returns a capability for the object to the location in CS space specified in the instruction.

e. Finally, the packet object has a space (IS) allocated for instructions. The instruction space would contain instructions for packet initialisation as well as instructions generated to represent Ada subprograms (procedures and functions). The index of entry points to various sections of the instruction space are stored in the HIA of the packet. In the case when a package in Ada is used for representing abstract data types, the IS of the packet would contain a section of the space allocated to instructions for creating an instance of the type. The entry point information for such a section would be available in the HIA (cf. section 4.6).

---

[5] It should be noted that the notion of 'own' variable in Ada arises only in case of variables in Ada packages and a section of the code in the package is responsible for initialisation of such variables.

The format of a packet object representation in the packet space of memory is shown in figure 4.9(a). It contains a fixed length section of five words called the packet header. The details of the header information area are shown in figure 4.9(b). A restriction in this design is that a packet could have 16 protected entry points. The relative address (relative to the end of the HIA) of the protected entry points are stored in the fields EPI1 to EPI16 in the Packet header. The remaining word of the header contains indexes to the four area in the packet (OVS, DTS, CS and IS) described earlier.

A packet object is created by execution of a CREATE PACKET instruction. The instruction is similar to the CREATE MODULE instruction in SWARD. When a process enters a

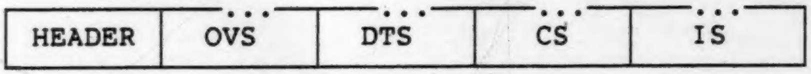


Figure 4.9(a): Representation of a Packet

1011	EP Index1	EP Index2	EP Index3	EP Index4
1001	EPI 5	EPI 6	EPI 7	EPI 8
1001	EPI 9	EPI 10	EPI 11	EPI 12
1001	EPI 13	EPI 14	EPI 15	EPI 15
1001	OVS Index	DTS Index	CS Index	IS Index

Figure 4.9(b): The Packet Header

a packet for execution, the capability register CR[9] gets loaded with the mapped capability for the packet object. Any address in the packet object is always evaluated with respect to CR[9]. Addresses, when specified in the instructions, are always relative to the beginning of the packet object.

During activation of a procedure, a function or a block in the packet, the local scalar variables are created on the activation stack by explicit instructions in the section of the IS containing code corresponding to the procedure, function or block. The descriptor templates or dope vectors are created on the activation stack by BMOVE instructions. The BMOVE instruction requires the starting and end of the source locations in the packet DCS space, the destination is automatically assumed to be starting from TOS. The majority of the instructions in the packet object would be stack oriented where the domain frame (corresponding to the packet) of the process stack segment is the implicit frame of reference. Thus the addresses of variables in the procedure, block or function are generated in the usual (lexical level, offset) form.

4.6 Implementation of Abstract Data Types

This section will deal with the support provided by the proposed architecture for implementation of Abstract data types(ADT).

When an Ada package defines an abstract data type, the spcification section of the package would contain the details of the representation of the type within the Private or Limited Private type declaration. The packet representation of such a package would contain a protected procedure (called INSTANTIATE) for creating an instance of the type in the heap space. It should also be noted that the subprogram or package that wants to use the private or limited private type for abstract data typing must explicitly specify the package (defining the abstract type) through a USE clause.

The mechanism of ADT implementation will be explained through an example. Let A be a packet object representing the package defining the abstract data type (say T). Let B be a packet representing a subprogram or package containing a declaration of a variable V of type T (i.e., V:T). The packet B would also have a capability (say C) for packet A in CS area of B (corresponding to the use clause in the subprogram or package and obtained by the CREATE PACKET instruction in the initialisation section of packet B).

The instructions in packet B corresponding to the elaboration of the declaration of the form V:T are as follows. To start with, a CREATE PACKET instruction would be executed that creates another instance of packet A, say A(V). The capability for A(V) (say C) would be returned and stored in the DCS area of B corresponding to the variable V. Now the entry point index for the INSTANTIATE procedure would be loaded on TOS and a COMPUTE CAPABILITY instruction would be executed with C as one of the operands. The capability C with appropriate index in the index/offset field would get deposited on TOS, on successful execution of the COMPUTE CAPABILITY instruction. The capability on TOS will be treated as an operand for the ENTER instruction that follows. The execution of the ENTER instruction would cause a domain switch. Then the INSTANTIATE procedure in packet A(V) would execute to create an instance of the ADT in the heap space and return a capability for the instance that would get stored in DCS area allocated to T in A(V).

To invoke any of the allowed operations on the variable V, two instructions are executed -

1) a COMPUTE CAPABILITY instruction with C and the EP Index of the desired operations as the two operands, and

2) an ENTER instruction with the capability on TOS (returned by the COMPUTE CAPABILITY instruction).

It should be noted that the protected procedure representing the specified operation (for the ADT) in packet A(V) would act on the representation in the heap space. The representation is pointed to by the capability of the heap object stored in the DCS area in A(V) corresponding to the variable T. The above description shows how the packet objects and tagged capabilities allow a straightforward implementation of abstract data types.

Chapter 5

The Instruction Set

The instruction set of the proposed architecture is summarized in this chapter. As indicated earlier this is a preliminary version of the instruction set and the instruction set is not complete without architectural support for tasking and input/output. The semantics of the proposed instructions are explained without any details of the exact formats and encoding.

The instructions are grouped according to commonality of functions. The groups are:

a. Stack group
b. Capability group
c. Branch group
d. Control group
e. Array group
f. Arithmetic-Logic group
g. Miscellaneous

Whenever necessary one or more S*A statements have been used for clarity of semantics.

5.1 The Stack Group

The common characteristic of this group of instructions is that the process stack top is implied to contain the source operand or the destination of the result of these instructions. The instructions in this group are further classified into four forms:

1. L - Form: the address is specified, in the 'lexical level, offset' form. The contents of CR[10],' the capability register associated with the stack segment is considered as the implicit base.
2. C - Form: the address is interpreted relative to a capability register specified in the instruction. There are two subforms for this form of instructions. Ca: the offset is specified in the instruction word; Cb: the offset is assumed to be on TOS.
3. P - Form: the address is interpreted relative to the top of the packet executing the instruction. The contents of CR[9] is the implicit base address for this form of instruction.
4. S - Form: the address is computed from the capability (of form '010' or '100') available either on the top of the stack (for load operations) or just below the data on the TOS (for store operations).

The form of the instruction is indicated within parentheses in the mnemonic for the instruction.

PUSH(L): The address of the operand is specified in the lexical level, offset form. The semantics could be represented as:

Stack [Sptr]: = Stack [DR[IR.LEX] + IR.offsett],

where any address in the 'Stack' is computed relative to

'The CR[10].base is implied to be the base address for effective address calculation for all the instructions in the group. Similarly the check against CR[10].limit is always performed.

CR[10].base. The effective address generated is checked against CR[10].limit for the relation --

effective address ≤ CR[10].base + CR[10].limit.

POP(L): The operand on TOS is stored in the stack location specified in the instruction. The corresponding S*A statement is:

Stack [DR[IR.Lex]+IR.offset]: = Stack [Sptr].

PUSA(L): The address specified by the Lex.level, offset form is pushed on top of the stack. The S*A representation is:

Stack [Sptr]: = DR [IR.Lex] + IR.offset;

PUSI: The operand specified in the instruction is pushed on the TOS.

PUSH(C): The operand address is computed relative to the 'base' of the capability register CR[i] (i=1...8) specified in the instruction. The 'offset' is specified in the instruction word for **PUSH(Ca)** and the offset is assumed to be on TOS for **PUSH(Cb)**. The S*A representation for PUSH(Ca) is:

Stack [Sptr]: = Mem[CR(i).base + IR.offset];

POP(C): The operand on TOS is stored in the effective address computed relative to the base of the capability register CR[i] specified in the instruction. It has two forms similar to the PUSH(C) instruction.

PUSH(P): The operand address is computed relative to the 'base' of CR[9], the capability register associated with the packet executing the instruction. The operand is loaded on TOS. There is no corresponding POP(P) instruction.

PUSH(S): The operand address is computed from a capability of the form '010' or '100' available on TOS. The semantics of the instruction implies invoking of the capability mapping mechanism to obtain the 'base', 'limit' information from the PMT. The corresponding POP operation is represented by POP(S).

Three operations that should be included in the instruction set are the increment, decrement and 'store zero' operations [Tan78]. The instructions are as follows:

INC(L): The operand address is specified by the lexical level, offset form of address specified in the instruction. The operand is expected to be an integer word. The operand is incremented by 1. The corresponding decrement operation is preformed by DEC(L) instruction. The INC and DEC operations are also available in form C.

STOZ(L,C): The instruction is available in forms L and C. On execution of this instruction, the operand specified by the effective address is set to zero.

CSTORE(L,C): (Check before storing): The instruction is available in both the forms. The address part of the instruction specifies the destination address for the ALU-output. The semantics of the instruction ensures checking of the ALU-output against the upper and lower range information available in the destination location. (The instruction was explained in Chapter 4).

LDVI, LDV2, LDV3: These three instructions are provided for loading of the three fields of the dope vector word. The offset is only specified in the instruction. The effective address of the dope vector word is computed relative to the contents of DR[Dis-top]. The operand is considered to be on the TOS. LDV1, LDV2 and LDV3 loads the 'upper bound', 'lower bound' and the 'length' fields of the dope vector word respectively.

BMOVE(P) OP1, OP2: (Block Move) The instruction is provided to move a block of words from the DCS area of the packet (executing the instruction) to the area on the stack segment, starting from the location pointed by the Sptr just prior to execution of this instruction. OP1 specifies the index in the DCS area and OP2 specifies the length of the block in words. The capability register CR[9] is implied in computing the effective address in the DCS area of the packet.

BCOPY ln, OP1, OP2: (Block Copy) The instruction is specifically meant for copying dope vectors between two different visible lexical levels. OP1 and OP2 represent the addresses of the source and destination dope vectors respectively. The addresses are specified in lexical level, offset forms. The parameter ln stands for the length of the dope vector in words. The instruction requires that the source and the destination locations have tags of '1010'. The instruction would only be used for passing array parameters. (The use of this instruction was further explained in Chapter 4).

ALLOC ln: This instruction is used to create objects in the heap space. It involves invocation of the heap manager. The structure to be created on the heap space is built on the temporary area of the latest activation record. The length information ln present in the instruction specifies the actual length of the representation in words. The heap manager would allocate the necessary space in the heap. The structure is copied onto the heap space and deallocated from the stack. The capability of form '001' is returned on TOS. A similar instruction is **PALLOC** ln, OP1. The difference between the two instructions is that the capability returned is stored in the DCS space of the packet object. The operand OP1 represents the offset in the DCS area and points to the capability space reserved in the template for the own variable under consideration.

5.2 The Capability Group

The instructions in this group are used for loading, storing capability registers and computing new capabilities of the forms '010', '011' and '000' from a capability of the form '001'. An instruction for modifying 'authority' field of a capability is also included in this group.

LCR i: (Load Capability register CR(i) for (i=1...8)). The instruction expects a capability word on the TOS. To start with, the capability on TOS is stored in the CRB area corresponding to CR(i) of the latest domain frame. Then the capability mapping mechanism is invoked. The mapped information is loaded into CR(i) along with the 'authority' field of the original capability and the L bit in the register is set.

SCR i: (Store Capability register CR(i) (for i=1...8)). The instruction loads the content of the CRB area, corresponding to CR(i) in the latest domain frame, and resets the L bit in CR(i). When the L bit in a capability register is not set, the content is invalid.

CCAP(L) OP1: (Compute Capability) In the L-form of the instruction, OP1 represents an address in the present domain. The instruction expects a capability of the form '001' in that address. If the authority field has 'enter' right, a capability of form '000' is produced and the 'index' field is loaded with the index from the TOS. If the authority field has other rights, then a capability of form '010' is produced with 'offset' field loaded from the TOS. Finally the capability is deposited on the TOS.

CCAP(P) OP1: The instruction is similar to the previous one except in that the effective address computed using OP1 represents a location in DCS/CS space having the desired capability.

5.3 The Branch Group

The instructions in the Branch group are used to branch within the instruction space of the packet. There are essentially two classes of instructions in this group: conditional branch and unconditional branch. For the mnemonics of the conditional branch instructions, the conditions are specified within parentheses.

The following instructions are in the class of conditional branch instruction.

CBRZ(=) OP1: The data on the TOS is an implicit operand. This data is compared for equality with zero. If the condition is satisfied, the program counter is incremented by the number of words specified in OP1. For any branch instruction, the effective branch address is checked against the 'base' + 'limit' information from CR[9]. The instructions CBRZ(≤), CBRZ(<), CBRZ(>), CBRZ(≥) and CBRZ(≠) are similar to the instruction explained above with different conditions for checking.

CBR(>) OP1: This conditional branch instruction assumes the contents of the top two locations of the stack as the two operands to be compared. If the contents of the TOS is greater than the operand below it, the branch takes place. Similar instruction for the other five conditions are available. These instructions are to be used for implementing IF and WHILE statements.

LOOP OP1 OP2: This instruction is provided for conditional looping over a set of instructions. The loop limit (i.e., TO part of a corresponding FOR instruction in Ada) is evaluated and pushed onto the TOS before executing this instruction. The offset of the loop variable in the stack is provided as the operand OP1, the forward branching offset in the IS of the packet is specified in OP2. The branch is executed if the loop variable equals the value in the TOS. A similar instruction is **RLOOP** where reverse checking is done for the loop variable and the loop limit.

The only other branch instruction is the unconditional branch. The mnemonic for the instruction is BR.

5.4 The Control Group

This group includes instructions for invocation of blocks, procedures and domain switching. All these instructions have been explained earlier in Chapters 3 and 4. A brief summary will be provided in this section.

CALL OP1: The instruction is always in form-L. The operand in the instruction is the address of the procedure identifier in lexical level, offset form. The procedure identifier is stored in the process stack at the lexical level of declaration of the procedure. The S*A description for the instruction was provided in Chapter 3 as the procedure M_2.CALL.

RETN: The return instruction causes return from a procedure to the instruction in the packet sequentially next to the CALL instruction. The semantics of the instruction is represented by the S*A description M_2.RETURN provided in Chapter 3.

PCALL OP1: The instruction is designed for calling procedure with parameters. The instruction would get executed only if the PB flag in the machine is set. The semantics of the instruction has already been described as the S*A description M_2.PCALL in Chapter 4.

PRETN: The instruction would be used for returning from a procedure called through a PCALL instruction. The S*A description for this instruction was given as M_2.PRETURN in Chapter 4.

FRETN: The instruction would be used for returning from a function call. The semantics is similar to RETN with a difference that before execution of FRETN the result of the function call is left in the TOS (specifically in the logically top-most register among the four registers representing the stack top). Any instruction following the FRETN would be able to obtain the result from the hardware register though the block representing the function body gets deallocated. Simple scalar results are available in the top-most register. For composite structures, the structure is allocated in the heap space before executing FRETN. The FRETN instruction loads the capability returned on the top of the stack to the logically top-most register in the group of four registers representing the top of stack for the following instruction.

The S*A descriptions for the instructions for block invocation and exit - **ENTRY, EXIT, PBEGIN,PEND** have already been provided in Chapters 3 and 4 (for mechanism M_2).

ENTER: This is a zero address instruction for domain switching. The instruction expects a capability of the form '000' with 'enter' authority. The execution of the process is transferred to the packet specified by the capability and the instruction execution starts at the entry point specified in the 'index' field of the capability. The instruction causes creation of a new domain frame on the process stack and the lexical level of the entered procedure is considered as 1. The details of the semantics of this complex instruction was provided in Chapter 4.

5.5 The Array Group

Three instructions are proposed in this thesis that deal with array structures.

DAS OP1, OP2: (Declare array space) The instruction is meant for creating the space for the array on the temporary area of the activation record and returns the base address of the space to the array base field in the dope vector. OP1 specifies the base address of the dope vector in lexical level, offset form and OP2 specifies the rank of the array.

INDEX OP1: The instruction computes the address of the actual array element using the dope vector address and subscripts available in sequence on the stack. The address is returned on the TOS after removal of the subscripts and the dope vector address from the stack. The instruction checks each and every subscript against the lower and upper bounds specified in the dope vector for the corresponding dimension. The operand OP1 specifies rank of the array.

SINDEX OP1: (Safe Index): The instruction is similar to the INDEX instruction but subscript bound-checking is not performed.

5.6 The Arithmetic Logic Group

The tagged memory representation allows generic arithmetic operations. The arithmetic instructions operate on the top two elements of the stack. The operands are popped off the stack and the result is pushed on the TOS. The instructions are ADD, SUB, MULT, DIV and NEG (changes the sign of the operand).

All the standard logical operations are available. The instructions are AND, OR, XOR, COMP, GT, LT, EQ, LSHFT, RSHFT.

5.7 The Miscellaneous Group

The instructions in this group are: EXCH, DUP, CI and CR.

EXCH	: (Exchange)	The top two elements of the stack are exchanged by this operation.
DUP	: (Duplicate)	The contents of TOS is duplicated and

pushed on to the stack.

CI	: (Convert to integer) A real operand on TOS is converted to an integer representation. The result replaces the operand.
CR	: (Convert to real) The integer operand is converted to real and the result replaces the operand.

Chapter 6

Conclusion

This thesis demonstrates how the architectural concepts of capability based addressing and stack oriented instruction sets may be combined to yield an execution environment congenial to execution of Ada programs. Two of the principal contributions of the new language Ada are its features for representing the concepts of abstract data types and information hiding. None of the commercially successful architectures provide any architectural features that support implementation of such advanced programming language concepts. On the other hand some of the architectures, (e.g., IAPX 432, IBM SWARD, IBM System 38) do provide architectural features for supporting these concepts, but the complexity of the mechanisms are far beyond the necessities of an essentially compile time bound language environment (like Ada).

One of the prime objectives of this research was to design an architecture that supports efficient execution of Ada programs and allows efficient implementation of the Packages and abstract data types in Ada. The stepwise systematic development of the architecture in the thesis was provided to demonstrate that the majority of the architectural features were introduced with the objective of providing efficient execution support for Ada programs. It should be obvious from the thesis that no unnecessarily complex mechanisms were introduced just for the sake of innovation. The capability based addressing was primarily adopted as the basic addressing mechanism to provide elegant support for implementing packages and abstract data types in Ada. The commonality of the objective for the design of the language Ada and the capability architecture for supporting development of large scale reliable software was indicated in Chapter 1.

The importance of efficient architectural support for variable addressing mechanism in a block structured language environment was established in Chapter 3. A major contribution of this thesis is the proposal of a methodology for choosing the implementation technique for exo-architectural components of a language directed architecture. Two new complexity measures were proposed that serve as a basis for this methodology. A new result was derived (in Chapter 3), using the complexity measures, that indicated the superiority of Rohl's mechanism over the other three mechanisms for variable addressing in Ada. The basic architectural support for variable addressing in the proposed architecture was designed using this methodology.

Instead of proposing yet another language-directed architecture for Algol-like languages, this thesis has concentrated on some of the special features of Ada that require efficient run time support.

The semantics of the parameter mechanism in Ada is quite different from the previous languages in its requirement for run time parameter type checking. The implementation of this mechanism on conventional architectures tend to become quite inefficient. The examination of an effort to develop an experimental Ada compiler on VAX 11/780 at CMU emphasizes the point [SeH80]. The provision of the new instructions (PBEGIN, PCALL, PEND and PRTN) in this architecture practically eliminates the problem.

Similarly introduction of new primitive data types, hardware support and special instructions CSTORE and LOAD RANGE significantly reduce the execution time overhead for dynamic constraint checking. The proposed architecture also provides support for representation and handling of dynamic arrays and discriminant records in Ada.

Myers has indicated that the IBM SWARD machine could be considered to be directed towards Ada [Mye82]. A study of the architecture reveals that a module object can represent a module in MODULA, but the architecture has no facilities for abstract data typing (as in Ada). Similarly the architecture has no equivalent concept of lexical level addressing in Ada. The architecture was not designed to provide any support for addressing free variables, thus it can not provide any support for subroutine management in Ada. Moreover the architecture uses the long tag approach and is specifically designed to support execution time bound languages. The execution of programs written in a strongly typed language like Ada will be unnecessarily slow on this architecture.

An architecture that is heavily publicized as a high-level Ada machine is the IAPX 432 from Intel. The characterization is inaccurate in the sense that it has few direct relationships to Ada [Mye82]. The architecture could be better characterized as an 'operating system machine'. This architecture is essentially a representation of the HYDRA operating system on silicon [LEV81]. It has excellent features for operating system support for memory management, process synchronization, scheduling, etc. The architecture has adopted the partitioned memory approach to capability architecture design. Hence, domain switching is not very efficient. Moveover, every CALL in this architecture is a domain switching operation. There are no capability registers for address translation, though it provides onchip associative memories for address translation. The architecture does not provide any architectural support for free variable addressing and support for automatic subroutine management (as CALL or PCALL instruction in this design).

As indicated earlier the proposed design in not complete in the true sense. The instruction set is not precisely defined in terms of formats and encoding of the fields. Moreover architectural supports for tasking and exception handling in Ada were not considered.

Bibliography

[AlW75] W. G. Alexander and D. B. Wortman, Static and Dynamic Charateristics of XPL Programs, *Computer 8*, 11 (1975), 41-46.

[BaC74] W. A. Barrett and J. D. Couch, Compiler Construction - Theory - and Practice, *Science Research Associates, INC., Chicago*, 1974.

[Bar81] D. W. Barron, Pascal- The Language and its Implementation, *John Wiley & Sons*, 1981.

[BiB80] J. M. Bishop and D. W. Barron, Procedure Calling and Structured Architecture, *The Computer Journal 23*, 2 (1980), 115-123.

[BiB81] J. M. Bishop and D. W. Barron, Principle of Descriptors, *The Computer Journal 24*, 3 (1981), 210-221.

[Bis80] J. M. Bishop, Effective Machine Descriptors for Ada, *Proc. of the ACM-SIGPLAN Notices Symp. on the Ada Programming Language*, Dec., 1980, 235-242.

[Cra79] H. G. Cragon, An Evaluation of Code Space Requirements and Performance of Various Architectures, *ACM-SIGARCH 7*, 5 (Feb. 1979), 5-21.

[DaO82] S. Dasgupta and M. Olafsson, Towards a Family of Languages for the Design and Specification of Computer Architectures, *Proc. of 9th Annual Symp. on Comp. Arch.*, 1982.

[Das81] S. Dasgupta, S*A : A Language for Describing Computer Architectures. In Hartenstien, Ed., *Computer Hardware Description Languages and their Applications, North Hollandd Publ., Amsterdam*, 1981.

[Das82a] S. Dasgupta, A Definition of the Architecture

Description Language S*A, *Dept. of Computer Science, Univ. of Southwestern Louisiana*, 1982.

[Das82b] S. Dasgupta, Computer Design and Description Languages. In Yovits, Ed., *Advances in Computer. Academic Press*, 1982.

[Das83a] S. Dasgupta, On Style in Computer Architecture, *Proc. Intl. Symp. on Computing Systems, New Orleans*, March,1983.

[Das83b] S. Dasgupta, On the Verification of Computer Architecture Using a Computer Description Language, *Proc. 10th Annual Symp. on Comp. Arch., Stockholm*, June 1983.

[Den68] P. J. Denning, The Working Set Model for Program Behaviour, *Comm. ACM 11*, 5 (1968), 323-333.

[Den80] T. D. Dennis, A Capability Architecture, *Ph.D. Thesis, Purdue Univ.*, 1980.

[Den82] D. E. R. Denning, Cryptography and Data Security, *Addison Wesley Publishing Company*, 1982.

[DeP82a] M. DePrycker, A Performance Analysis of the Implementation of Addressing Methods in Block-Structured Languages, *IEEE Trans. on Computers C-31*, 2 (Feb. 1982), 155-163.

[DeP82b] M. DePrycker, On the Development of a Measurement System for HLL Program Statistics, *IEEE Trans. on Computers C31*, 9 (Sept. 1982), .

[DeV66] J. B. Dennis and E. C. VanHorn, Programming Semantics for Multiprogrammed Computations, *Comm. ACM 9*, 3 (March 1966), 143-155.

[DiS79] W. B. Dietz and L. Szewerenko, Architectural Efficiency Measures: An Overview of Three Studies, *Computer 12*, (Apr. 1979), 26-33.

[Dij60] E. W. Dijkstra, Recursive Programming, *Numer. Math. 2*, (1960), 312-318.

[Dij68] E. W. Dijkstra, The Structure of THE Multiprogramming System, *Comm. ACM 11*, 5 (May 1968), 341-346.

[DiM82] D. Ditzel and R. McLellan, Register Allocation for Free : The C-Machine Stack Cache, *Proceedings : Symp. on Arch. Support for Prog. Languages and Operating Systems*, March, 1982.

[DiP80] D. R. Ditzel and D. A. Patterson, Retrospective on High-level Language Computer Architecture, *ACM-SIGARCH 8*, 3 (1980), 97-104.

[Dit80] D. R. Ditzel, Investigation of a High Level Language Oriented Computer for X-Tree, *Bell Labs., Computing Sc. Tech. Report No. 88*, Nov. 1980.

[Dom80] O. Dommergard, The Design of a Virtual Machine for Ada. In Bjorner & Oest, Ed., *Towards a Formal Description of Ada. Lecture Notes in Comp. Sc. 98*, (1980), .

[Dor79] R. W. Doran, Computer Architecture : A Structured Approach, *Academic Press*, 1979.

[Els76] J. L. Elshoff, An Analysis of Some Commercial PL/I Programs, *IEEE Trans. on Software Engg. SE-2*, 2 (1976), 113-120.

[Eng72] D. M. England, Architectural Features of System 250, *Infotech State of the Art Report 14 : Operating Systems*, 1972, 395-428.

[Fab74] R. S. Fabry, Capability-Based Addressing, *Comm. ACM 17*, 7 (July 1974), 403-412.

[FuB77] S. H. Fuller and W. E. Burr, Measurement and Evaluation of Alternative Computer Architectures, *Computer 10*, (Oct. 1977), 24-35.

[Feu73] E. A. Feustel, On the Advantages of Tagged Architecture, *IEEE Trans. on Computers C-22*, 7 (JULY 1973), 644-656.

[FlH83] M. J. Flynn and L. W. Hoevel, Execution Architecture: The Deltran Experiment, *IEEE Trans. on Computers C32*, 2 (Feb. 1983), 155-163.

[Geh79] E. F. Gehringer, Variable Length Capabilities as a Solution to the Small-Object Problem, *Proc. of the 7th Symp. on Operating Systems Principles, NY: ACM*, 1979, 131-142.

[GoH80] G. Goos and J. Hartmanis, The Programming Language Ada (Reference Manual), *Lecture Notes in Comp. Sc. 106, Springer Verlog*, 1980.

[Gli79] V. D. Gligor, Architectural Implications of Abstract Data Type Implementation, *6th Annual Symp. on Computer Architecture*, 1979, 21-30.

[GrD72] G. S. Graham and P. J. Denning, Protection-Principles and Practice, *Spring Joint Computer*

Conference, 1972, 417-429.

[Gri71] D. Gries, Compiler Construction for Digital Computers, *John Wiley & Sons*, 1971.

[Gri74] M. Griffiths, Run-time Storage Management, *Lecture Notes in Computer Science, Springer-Verlag, New York*, 1974.

[Hil81] O. D. Hill, A Hardware Mechanism for Supporting Range Checks, *ACM-SIGARCH 9*, 4 (1981), 15-21.

[Hor83] E. Horowitz, Fundamentals of Programming Languages, *Computer Science Press*, 1983.

[Jag80] A. Jagannathan, A Technique for the Architectural Implementation of Software Subsystems, *ACM- Annual Symp. on Comp. Architectures*, 1980, 236-244.

[JeW72] K. Jensen and N. Wirth, Pascal User Manual and Report, *Springer, Berlin*, 1972.

[Kee78a] J. L. Keedy, On the Use of Stacks in the Evaluation of Expressions, *ACM-SIGARCH 6*, 6 (Feb. 1978), 22-28.

[Kee78b] J. L. Keedy, On the Evaluation of Expressions Using Accumulators, Stacks and Store-to-Store Instructions, *ACM-SIGARCH 7*, 4 (Dec. 1978), 24-27.

[Kee79a] J. L. Keedy, More on the Use of Stacks in the Evaluation of Expressions, *ACM-SIGARCH 7*, 6 (JUNE 1979), 18-21.

[Kee79b] J. L. Keedy, A Technique for Passing Reference Parameters in an Information Hiding Architecture, *Comuter Architecture News 7*, 9 (1979), 11-15.

[LaS76] B. W. Lampson and H. E. Sturgis, Reflections on an Operating System Design, *Comm. ACM 19*, 5 (May 1976), 251-265.

[Lev81] H. M. Levy, A Comparative Study of Capability Based Architectures, *Tech. Report 81-12-01, Dept. of Comp. Sc., Univ. of Washington*, Dec.. 1981.

[Lin76] T. A. Linden, Operating System Structures to Support Security and Reliable Software, *Computing Surveys 8*, 4 (Dec. 1976), 409-435.

[MaD74] S. E. Madnick and J. J. Donovan, Operating Systems, *McGraw Hill Book Co.*, 1974.

[MyB80] G. J. Myers and B. R. S. Buckingham, A Hardware

Implementation of Capability-Based Addressing, *Operating Systems Review 14*, 4 (Oct 1980), 13-25.

[Mye77] G. J. Myers, The Case Against Stack Oriented Instruction Sets, *ACM-SIGARCH 6*, 3 (Aug. 1977), 7-10.

[Mye78] G. J. Myers, The Evaluation of Expressions in a Storage-to-Storage Architechture, *ACM-SIGARCH 6*, 9 (SEPT 1978), 20-23.

[Mye82] G. J. Myers, Advances in Computer Architecture, *John Wiley & Sons*, 1982.

[NeW77] R. M. Needham and R. D. H. Walker, Cambridge CAP Computer and its Protection System, *Proceof sixth ACM Symposium on Operating Systems Principles*, Nov. 1977, 1-10.

[NeW74] R. M. Needham and M. V. Wilkes, Domains of protection and the management of processes, *The Computer Journal 17*, 2 (1974), 117-120.

[RaL81] J. Rattner and W. W. Lattin, Ada Determines Architecture of 32-bit Microprocessor, *Electronics 54*, 4 (Feb., 1981), .

[Ree80] M. J. Rees, Pascal on an Advanced Architecture, In Barron Ed., *Pascal- The Language and its Implementation. Wiley, London*, 1980.

[RaR64] B. Randell and L. J. Russel, ALGOL 60 Implementation, *Academic Press, New York*, 1964.

[Roh75] J. S. Rohl, An Introduction to Compiler Writing, *MacDonald & Jane's, London*, 1975.

[RoT76] S. K. Robinson and I. S. Torsun, An Empirical Study of FORTRAN Programs, *Computer Journal 19*, 1 (1976), 56-62.

[SaS75] J. H. Saltzer and M. D. Schroeder, The Protection of Information in Computer System, *Proceedings of the IEEE 63*, 9 (September 1975), 1278-1308.

[ShH80] M. Sherman and A. Hisgen, An Ada Code Generator for VAX 11/780 with UNIX, *Proc. ACM-SIGPLAN Notices Symp. on the Ada Prog. Language, Boston*, 1980, 91-100.

[Sim69] H. A. Simon, The Science of The Artificial, *MIT Press, Cambridge, Mass.*, 1969.

[Sto80] H. S. Stone, Introduction to Computer Architecture, *Second Edition, Scince Research Assoc. Inc.*, 1980.

[Tan76] A. S. Tanenbaum, Structured Computer Organization, *Prentice Hall, INC.*, 1976.

[Tan78] A. S. Tanenbaum, Implications of Structured Programming for Machine Architecture, *Comm. ACM 21*, 3 (1978), 237-246.

[Wel76] T. A. Welch, An Investigation of Descriptor Oriented Architecture, *The 3rd Annual Symp. on Comp. Architecture*, 1976, 141-146.

[Wil72] M. Wilkes, Time Sharing Computer Systems, *New York : American Elsevier, 2nd ed.*, 1972.

[Wil82] M. V. Wilkes, Hardware Support for Memory Protection, *Symp. on Architectural Support for Prog. Languages and Operating Systems*, March, 1982, 107-116.

[Wor72] D. B. Wortman, A Study of Language Directed Computer Design, *Ph.D. Dissertation, Stanford University*, 1972.

[Zei81] S. Zeigler, Ada for the Intel 432 Computer, *IEEE Computer*, June, 1981.